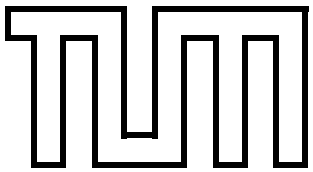

The activity support system TACTS

Gunnar Teege

TUM–I9306
April 1993

Contents

1 Introduction **1**

2 The ELIOOS Object-Oriented System **4**
2.1 Overview 4
2.1.1 The Object Base 4
2.1.2 The Type Hierarchy 5
2.1.3 Methods 7
2.2 The Implementation 9
2.2.1 The Implementation Environment 9
2.2.2 The User Interface 9

3 Basic Mechanisms **11**
3.1 Attributes 11
3.1.1 User-editable String Attributes 11
3.1.2 Descriptions 12
3.1.3 References 12
3.2 Collections 12
3.2.1 Object Collections 12
3.2.2 Displaying and Member Lists for User Interaction 13
3.2.3 Other Operations on Collections 15
3.2.4 Activity sets 15
3.3 Time Specifications 16
3.3.1 Interval Sets and Operations 16
3.3.2 Operations on Timed Objects 18
3.3.3 Combining Time Specification and Collections 18
3.4 State Management 19
3.4.1 Automata as Type Constituents 20
3.4.2 Operations on Objects with State 20
3.4.3 Starting and Finishing Objects 21
3.4.4 Combining State and Collections 22
3.4.5 Combining State and Time Specification 23
3.4.6 Executing a Transition upon Instance Creation 24
3.4.7 Refined Transitions 24

3.4.8 Programming of Execution Plans with Refined Transitions 25

4 The Core of TACTS **27**
4.1 Activities . . . 27
4.1.1 The Basic Automata . . . 27
4.1.2 Basic Activities and Structured Activities . . . 28
4.2 Agents . . . 29
4.2.1 Agent Objects . . . 29
4.2.2 Agent Sets . . . 30
4.2.3 Activities with Agents . . . 30
4.3 The Executing Agent of an Activity . . . 31
4.3.1 Activities Executed by a Single Agent . . . 31
4.3.2 Activities Executed by the User . . . 31
4.4 The Partner Agent of an Activity . . . 32
4.4.1 Specifying an Activity Partner . . . 32
4.4.2 Using the Partner Specification . . . 32
4.5 Basic Communication . . . 32
4.5.1 Information Transmission . . . 32
4.5.2 Communication with an Activity Partner . . . 33

5 Requests **35**
5.1 Classes for Team Activities . . . 35
5.1.1 Team Activities . . . 35
5.1.2 Virtual User Concept Types . . . 36
5.1.3 The Virtual Type for Requests . . . 36
5.2 Fixing a Role in Team Actions . . . 37
5.2.1 Role-Specific Types . . . 38
5.2.2 The Role-Specific Types for Requests . . . 38

6 Documents **40**
6.1 Abstract Documents . . . 40
6.1.1 Operations on Abstract Documents . . . 41
6.1.2 The States of Abstract Documents . . . 42
6.2 Document History . . . 43
6.2.1 Modeling Document Operations by Subactivities . . . 43
6.2.2 The Subactivities for the Document Methods . . . 44
6.3 Document Versions . . . 44
6.3.1 Definition of Document Versions . . . 44
6.3.2 Operations for Document Versions . . . 45
6.3.3 Named Versions . . . 45
6.4 Document Structure . . . 46
6.4.1 Simple Documents . . . 46
6.4.2 Structured Documents . . . 47

6.4.3 Homogeneously Structured Documents 48
6.4.4 Combination with Version Management 49
6.5 Document Content 50
6.5.1 Internal Documents 51
6.5.2 Filed Documents 52
6.5.3 Filed Documents and Structure 52
6.5.4 Document Content Kinds 53
6.6 Processable Documents 53
6.6.1 One-Step Processable Documents 55
6.6.2 History and Versions for Processable Documents 56
6.6.3 TEX Documents 56

7 Communication **59**
7.1 Supporting Communication 59
7.1.1 Communication Environments and Addressing 59
7.1.2 The Transmission of Text Pieces with Headers 60
7.1.3 The Interface to the Transmission Medium 61
7.1.4 The Interface to the Transmitted Object 62
7.1.5 Communication and Document History 63
7.1.6 Communication and TEX Documents 64
7.2 Communication Activities 64
7.2.1 Activities for Document Transmission 64
7.2.2 Header Management for Transmission Activities 66
7.2.3 Sending and Receiving Mail and News 67

8 Conclusion **70**

Chapter 1

Introduction

In the context of CSCW-systems, several systems have been developed, which try to support collaborative work. Among others, there are systems for the support of

- communication and coordination of the activities of a group of people, like the Coordinator [11], or the Strudel toolkit [10],
- office procedures like the Domino System [2], or the Prominand System [4],
- collaborative editing of documents, like Prep [8], or Quilt [1],
- other aspects of collaborative work, like meeting scheduling and project management.

However, all these systems only attack one aspect or a few aspects of collaborative work. Typically, they use only one paradigm, such as procedure-orientation, conversation-orientation, or form-orientation. The TACTS system, instead, tries to integrate many aspects and paradigms in one system. Of course, it can do so only incrementally, by being a frame system that allows the integration of different aspects and paradigms. The main requirements to a system of this kind are flexibility and extendibility. Consequently, TACTS is designed as an environment for the experimental design of a toolkit for supporting collaborative work.

The importance of flexibility, extendibility, and integration is also emphasized by Kreifelts in his report about the experiences with the Domino system [5]. Among other deficiencies, it was found that the exception handling facilities were not flexible enough, mechanisms for the communication of informal information were missing, and there was a lack of integration of other tools, like electronic mail or spreadsheets.

As a general design principle, TACTS uses activities of a single system user as central basic units. This approach is of a finer grain than that of most other systems which use group activities [7] or tasks with contexts [2] as basic building blocks. The TACTS approach enables the modeling of all these complex units

by constructing them starting with the basic units. The TACTS model includes three basic types of activity support.

Structuring: Managing a structure of all activities of the user and of relations between them.

Protocol: Protocoling the activities of the user and storing, e.g., the time and outcome of their execution.

Execution: Executing the activities. If possible, the system executes parts or all of the activity itself, or it activates corresponding tools.

The reasons for calling TACTS an *activity support system* are the activity being a basic unit together with the three basic support types.

To fully define and understand an activity, at least a minimal context is required. The philosophical framework of Activity Theory describes activities in this way [6]. The main parts of the context are the object which is manipulated, the active subject which performs the activity, and a community which is involved. Accordingly, TACTS supports the representation of other units aside from activities. Examples are documents which are objects manipulated during activities, or agents which represent active or passive subjects. However, TACTS is not restricted to a fixed set of units. Any kind of unit with any semantics can be introduced and used in the system.

The approach of TACTS has some interesting consequences. Since the basic units model activities of a single user and are local to that user, it is possible for some members of a group to use TACTS, while others don't. Systems using a centralized approach to group work, where the basic unit is a group activity, do not allow this flexibility. With TACTS, every user may decide whether to use TACTS, and when used, for which projects. With centralized systems, all group members must use the system, if the group work is supported by the system. Of course, in TACTS additional support can be provided if all members of a group use the system.

Another consequence is that TACTS supports activities which do not involve collaborative work, as well. Many collaborative activities have subordinate non-collaborative activities, such as writing a short report as part of a larger project involving other people. TACTS treats collaborative and non-collaborative activities in a uniform way. Both may be supported by structuring, protocoling, and execution.

This does not mean that higher level aspects of collaborative work can not be represented in TACTS. Medina-Mora distinguishes in [7] the level of information processes and the level of business processes, where the former describes the way information is passed, and the latter describes the reason, expectations and dependencies of information passing. In TACTS both levels may be represented in a common framework, the business level being implemented on top of the information level.

TACTS is an evolving system. Some aspects are already included in the current version. Among them are office procedures, group interaction, document development, time planning, and electronic communication. All of them still can be strongly improved and extended, and additional aspects can be added. One important point is the extension of the system by the end user. Kuutti argues in [6] that a system for activity support should not only support performing the activity instances. Instead, it should support the design of tools, rules, and schemes which help in performing activity instances. As an example, the system should support the definition and manipulation of procedure execution plans, not only the execution of a number of fixed plans.

To achieve the required flexibility and extendibility, TACTS uses an object-oriented approach. Multiple inheritance is used massively to allow the combination of properties and aspects of objects in an easy way. There are two levels of configuring the system to the user's needs. Upon creation of an object instance in the system, the user may determine the properties of the instance by selecting one or more types from the set of predefined types. Additional types may even be added later, during the instance's lifetime. Aside from this, the user may define his/her own types by combining and specializing existing types. Thus the user may design own tools for supporting activity instances. The support for user defined types in TACTS is still rudimentary, and will be improved. However, currently it is possible to specify procedure execution plans by defining corresponding types.

The approach benefits from several specific mechanisms in the object-oriented system and from a specific methodology for designing the type hierarchy. TACTS is implemented on top of the object-oriented system ELIOOS which was specifically developed to meet the requirements. The type hierarchy is designed using and extending the principle of *fine-grain inheritance*, described in [3].

This report contains a detailed description of the current type hierarchy implemented in TACTS. It is not meant to be a user manual for TACTS. Instead, it should serve the following goals. First, it should be a detailed description of the internal structure of the current TACTS version. Second, it should demonstrate the principle of applying fine-grain inheritance for the construction of a flexible and extendible toolkit for activity support. Third, it should give examples for the realization and integration of several different mechanisms in the object-oriented framework. Finally, it should be the starting point for the evolution and extension of the TACTS system.

We would like to thank Johann Schlichter and Uwe M. Borghoff for valuable contributions to the report. Many thanks are also due to Franz Vojik, Friedl Bunke, Armin Liebl, and Christof A. Neumann for many stimulating discussions regarding the TACTS design.

Chapter 2

The ELIOOS Object-Oriented System

TACTS is implemented on top of ELIOOS, the Emacs Lisp Object-Oriented System. ELIOOS was specially developed for TACTS, however, it can support other systems as well.

2.1 Overview

In this section we give an overview of those features of ELIOOS which are crucial for the functionality of TACTS. This is not meant to be a full documentation of ELIOOS, it should only give a feeling.

2.1.1 The Object Base

ELIOOS maintains a set of objects, called the *object base*. The structure of a single object is that of an attribute–value list. Every list element consists of an attribute name and an arbitrary string as attribute value.

The object set is structured on three levels. On the lowest level objects form *groups*. A group may contain an arbitrary number of objects. Each object must belong to exactly one group. Several groups together form a *base*. Every base may be loaded separately into the system. All bases of one user form together a *user space*. Each user has only access to the objects of his/her own user space. Thus, the objects are local to the user.

Every object has a unique name which is fixed during the object's lifetime and is not reused after the object's deletion. The object name may be used to access the attribute list associated with the object as long as the object exists. Thus the name can be used for referencing an object from other objects. Every object is created in a certain group and base of a user space. The user can delete it or move it into other groups or bases in the same user space.

2.1.2 The Type Hierarchy

The common properties of a class of objects are defined and implemented by a *type*. A type is simply a collection of items which describe object properties in some way. One example for these items are methods.

There are two ways of specifying a type: by explicitly defining type constituents or by combining other types. Normally these mechanisms are mixed when a type is defined. One or more existing types are combined and extended by explicit definitions of constituents. When types with common constituents are combined, these constituents are combined in a special way, resulting in true multiple inheritance.

Types combined in one step have no special order which effects the combination of constituents[1]. The resulting type, however, is more special than any of the types used in the combination. This defines a partial ordering on all types and results in the usual class hierarchy.

Additionally, there are mechanisms for explicitly specifying automatic specialization and incompatibility. Automatic specialization automatically adds some specified types whenever a certain set of types is combined. For every set of types additional types may be specified for automatic specialization. The typical application is to add certain properties to handle the combination of properties of two types.

Any set of types may be declared incompatible. Incompatible types may not be combined. The incompatibility applies only to the set as a whole. True subsets may be combined as long as there is no explicit incompatibility declaration for the subset itself. Note that types are never incompatible due to their definition. Any two types may be combined if they are not explicitly declared incompatible.

Every object in the object base belongs to a type. Upon creation of an object the user may select types from a menu of type names. The selected types are dynamically combined and the result is used as type for the object. The type of an existing object can be specialized at any time by combining it with additional types. However, it is not possible to generalize it by "removing types".

In the menus dependencies between types are made visible to the user in the following ways. After every selection of a type by the user all types which are incompatible with the set of types selected so far are removed from the menu. Additionally, all supertypes of selected types are automatically marked as selected as well. Figure 2.1 depicts the menu for the activity types in TACTS (see Chapter 4) before and after selecting the type **by-partner**. On the right hand side it can be seen that the entries for the types **inherit-by** and **by-self** are removed (since they are incompatible with **by-partner**) and that the entries for the types **by-single** and **init-by** are marked as well (since they are supertypes

[1] In other systems the type order is often used to resolve conflicts created by multiple inheritance. In ELIOOS another approach is used that handles multiple inheritance in a much more flexible way.

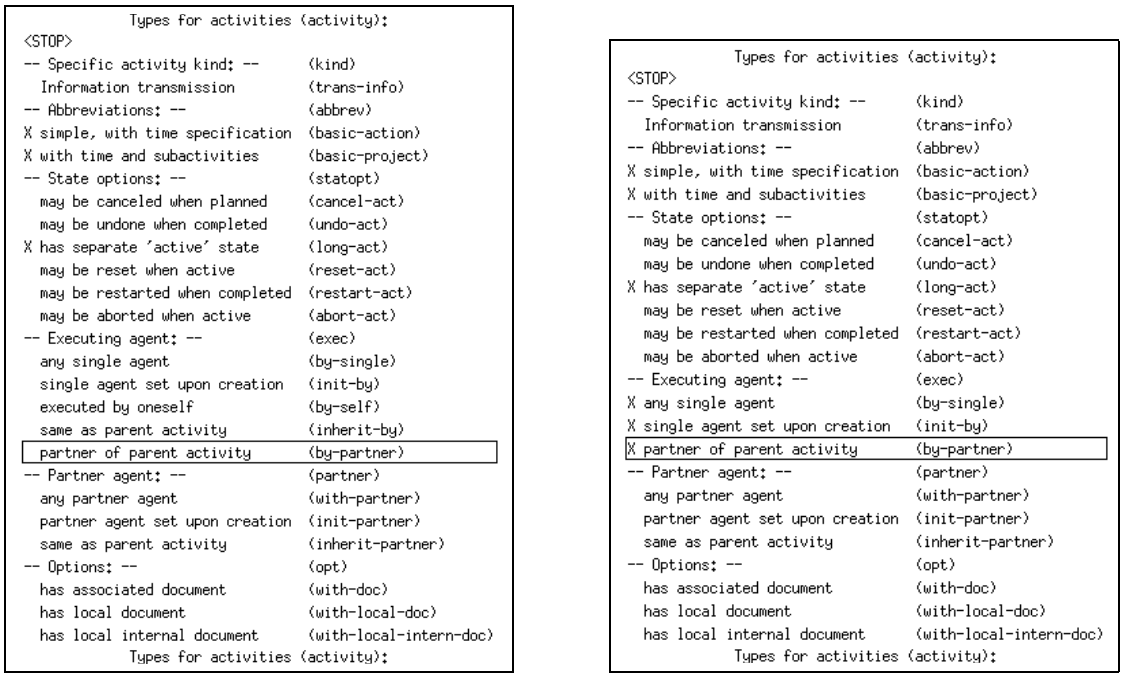
Types for activities (activity):
<STOP>
-- Specific activity kind: -- (kind)
Information transmission (trans-info)
-- Abbreviations: -- (abbrev)
X simple, with time specification (basic-action)
X with time and subactivities (basic-project)
-- State options: -- (statopt)
may be canceled when planned (cancel-act)
may be undone when completed (undo-act)
X has separate 'active' state (long-act)
may be reset when active (reset-act)
may be restarted when completed (restart-act)
may be aborted when active (abort-act)
-- Executing agent: -- (exec)
any single agent (by-single)
single agent set upon creation (init-by)
executed by oneself (by-self)
same as parent activity (inherit-by)
partner of parent activity (by-partner)
-- Partner agent: -- (partner)
any partner agent (with-partner)
partner agent set upon creation (init-partner)
same as parent activity (inherit-partner)
-- Options: -- (opt)
has associated document (with-doc)
has local document (with-local-doc)
has local internal document (with-local-intern-doc)
Types for activities (activity):
Types for activities (activity):
<STOP>
-- Specific activity kind: -- (kind)
Information transmission (trans-info)
-- Abbreviations: -- (abbrev)
X simple, with time specification (basic-action)
X with time and subactivities (basic-project)
-- State options: -- (statopt)
may be canceled when planned (cancel-act)
may be undone when completed (undo-act)
X has separate 'active' state (long-act)
may be reset when active (reset-act)
may be restarted when completed (restart-act)
may be aborted when active (abort-act)
-- Executing agent: -- (exec)
X any single agent (by-single)
X single agent set upon creation (init-by)
X partner of parent activity (by-partner)
-- Partner agent: -- (partner)
any partner agent (with-partner)
partner agent set upon creation (init-partner)
same as parent activity (inherit-partner)
-- Options: -- (opt)
has associated document (with-doc)
has local document (with-local-doc)
has local internal document (with-local-intern-doc)
Types for activities (activity):

Figure 2.1: Selecting a Type in ELIOOS

of type **by-partner**).

The selection of types by the user and their combination at run time is most effective, if the type hierarchy is designed accordingly. It should offer maximal flexibility to the user. To do this, two principles should be respected in the design of the type hierarchy. First, every type should only contain one single concept or aspect. This is the principle of fine-grain inheritance, described in [3]. Second, for all types which are not semantically incompatible, it should be possible to combine them, yielding a type which combines the concepts in a reasonable way.

These principles motivate the following informal classification of types.

- A *user concept* is a type containing a single concept. It may be selected by the user to create or specialize an instance. A user concept may have other user concepts as supertypes, if these concepts are necessary requirements for the new concept. Thus, in general the user concepts form a hierarchy which is a part of the hierarchy of all types.

- A *user packet* is a type which is equivalent to the combination of certain user concepts. It may be selected by the user and is offered for convenience reasons. User packets are useful for frequently needed user concept combinations.

- A *designer concept* is a type containing a single concept. However, it is only accessible to the type hierarchy designer, not to the user. Designer concepts

are useful for basic mechanisms which need additional specialization before they can be used in an instance.

- A *combination supplement* is a type which contains additional features that are needed to make a combination of other types behave in the expected way. Combination supplements are not visible to the user. They are added to type combinations by automatic specialization.

In TACTS, a simple syntactic convention is used for distinguishing user concepts and user packets from other types. The type names for designer concepts start with a star "*", those for combination supplements start with a percentage sign "%".

2.1.3 Methods

The most important sort of type constituent (and the only one predefined in ELIOOS) is the *method*. Methods, as usual, define operations on objects. Methods associated with type T are activated in the context of individual objects of type T. A method may have arbitrary many parameters and it may have a result.

To allow for method combination during multiple inheritance, all method parameters are position independent and must be supplied by (name, value) pairs. The method body is a sequence of named parts which are executed one after the other. The parts may have position constraints ("before" or "after") relative to parts of the same method of other types. Upon combination the part sequences are merged in a way to meet these constraints. Thus, methods may not only be specialized by adding actions to be executed before or after the whole method. Instead, additional actions may be inserted before or after any method part.

For documentation purposes every method has a documentation string. It should describe the actions invoked by the method parts. Upon combination, all documentation strings are concatenated to be compatible to the partial order of the types. Thus even methods constructed by dynamic combination have a documentation.

There are some standard methods which are called (but not defined) by the ELIOOS system itself. The most important of them is the method **created**. It is called immediately after the creation of an object and should initialize the object, e.g., create and initialize attributes of the object. The method **delete** is called immediately before an object is deleted, the method **moved** is called immediately after an object has been moved.

Whenever the type of an object is specialized by adding types, the new parts of the method **created** are executed, i.e., all parts not belonging to the old types' method **created**. Thus, e.g., additional attributes may be created and initialized. ELIOOS implements this kind of "method subtraction" for arbitrary methods,

not only for the method **created**. However, normally it does not make sense to apply it to other methods. Subtracting a method A from a method B is defined to result in a method C. If C and A are combined, this yields the method B again. Additionally, C may not have parts in common with A.

Method combination is the main mechanism for implementing multiple inheritance in ELIOOS. The combination of n methods contains all parts belonging to the single methods. However, parts which are inherited along several different paths in the type hierarchy are only contained once. Attributes are introduced by parts of the method **created** and are thus inherited in the same way. The following example should illustrate multiple inheritance by method combination in ELIOOS. We use an informal syntax for defining types, methods, and method parts. The meaning should be apparent.

```
typeA:
        created:
                partA1: create attribute X

typeB: specialization of typeA
        created:
                partB1: create attribute Y
                partB2: /after partA1/ set attribute X to VAL1

typeC: specialization of typeA
        created:
                partC1: /before partA1/ create attribute Y
                partC2: create attribute Z

typeD: combination of typeB and typeC
```

One possible form of the method **created** of type **typeD** is the following:

```
partC1
partA1
partB1
partC2
partB2
```

Note that partA1 appears only once, although it is inherited via **typeB** and **typeC**. The execution of this method creates the attributes Y, X, and Z and sets X to VAL1. The creation of Y is done twice, because the corresponding steps are inherited from different types. In this case the second creation is ignored. Thus ELIOOS implements attributes with the same name inherited from different types by a single common attribute of this name.

2.2 The Implementation

Now we present some aspects of the ELIOOS implementation. These may be of interest to the user, since they involve the basic layers of the user interface.

2.2.1 The Implementation Environment

For several reasons we chose GNU Emacs as the implementation environment. First, an Emacs lisp program is highly portable. Even the compiled form can be transferred directly to any machine operating GNU Emacs. Second, Emacs lisp allows direct access to the multi-buffer / multi-window user interface of Emacs, to text editing facilities, to file handling facilities and to subsystems for sending and receiving electronic mail or for using the USENET news system. Furthermore there is an interface to the operating system which allows the use of all system commands. Thus it is possible to access printers, compilers, or text processing systems. Third, emacs provides a comfortable programming and documenting environment which includes a symbolic debugger.

The ELIOOS system is simply a set of emacs lisp functions and variables. It mainly implements the object base management, the type hierarchy management and a basic user interface. Currently, every base in a user space is stored in a separate text file using a special format. There are functions for loading and parsing these files. The user does not work directly with base files. The type hierarchy is defined in lisp program files using a special syntax implemented by lisp macros. The hierarchy is loaded when the user starts to work with ELIOOS. At any time it is possible to load the definitions of additional types.

2.2.2 The User Interface

The main user interface is a group of "object buffers". These are emacs buffers with an attached object from a loaded base. Normally, an object attaches itself to an object buffer when the method **attach** is called for it. Additionally, during the execution of this method the object may display some information about itself in the buffer.

The system includes additional buffers, e.g., buffers for displaying or editing text. One use of these buffers is the direct access to attribute lists of objects for debugging purposes.

Associated with an object buffer are a number of key combinations for issuing special commands. Some of these commands apply to the attached object, e.g., the object can be deleted, specialized, or a method can be activated for it. If the method has parameters, their values may be entered interactively using the emacs minibuffer. Depending on the type of the attached object there may be additional keys defined which activate methods defined for the attached object.

Other commands apply to the base of the attached object, or to the ELIOOS system as a whole. It is possible to create, load, or save a base and to switch between several loaded bases. Finally, the user can enter the normal emacs commands for switching between buffers, or resizing or deleting windows.

When GNU Emacs runs under the X Window System, it supports mouse clicks and popup menus. These are used in TACTS in several ways to provide an alternative interface. One example is method activation by the mouse. A menu is popped up displaying the list of methods applicable to the object (depending on its type). The method to be activated can then be selected with a simple mouse click.

Chapter 3

Basic Mechanisms

This chapter describes the basic mechanisms of TACTS. They are implemented as ELIOOS types and can be used separately or in combination.

3.1 Attributes

In ELIOOS, attributes are the single means to specify information for instances. In this section we introduce several specific application kinds for attributes used in TACTS. The corresponding types either define specific attributes, or they define methods which can be used to manipulate attributes of a certain kind. In the first case the type is is a user concept, in the second case it is a designer concept which needs additional specialization (see Section 2.1.2).

3.1.1 User-editable String Attributes

It is often useful to associate one or more short text strings with an object. Examples are a name for an agent object, or a medium specification for an information transmission object. In ELIOOS, all attribute values are strings. Thus attributes may directly be used to hold strings associated with an object. However, mechanisms must be added to display the string and to allow interactive editing.

These mechanisms are defined in type ***free**. It is the root type of the TACTS type hierarchy. Since it is a supertype of every other TACTS type, every TACTS type may define user-editable attributes. The type ***free** introduces two new methods: **display** and **set-attribute**. The method **display** is used to display information about the object. Whenever a new string attribute is defined in a type, the method **display** must be enhanced (a new method part must be defined) to display the new attribute's value, if this is required. The method **set-attribute** may be activated by the user to change the value of a given string attribute interactively.

The type ***free** introduces one string attribute, the *shortname*. It can be used to provide a short characterization about the object. In some cases it is used for user input of object references. However, in general it need not be unique and it is not required to be set for all objects.

3.1.2 Descriptions

A special case of a string associated with an object is a *description* of the object. The type **with-descr** defines a multi-line description using the attribute **Descr**. The difference between this attribute and normal string attributes, as described in Section 3.1.1, is the fact that the special method **edit-descr** may be used to change a description. This methods supplies better support for editing a multi-line string than **set-attribute**. However, an object may have only one description of this kind.

The type **with-descr** is a user concept. The user decides which objects have a description by adding the type **with-descr** to the objects' types either upon creation of the object or later during a specialization of the object.

3.1.3 References

An attribute may be used to hold a reference to another object. This is implemented by storing the object name (see Section 2.1.1) in the attribute. The type ***with-ref** defines three methods which support attributes with object references. The method **set-ref** can be used to set or change an object reference. The method **deref** tries to dereference a given reference attribute. The method **follow-ref** displays information about the object referenced by a given attribute, using the mechanism introduced in Section 3.2.2.

The type ***with-ref** is a designer concept. It does not make sense to instantiate it directly, since it does not define an attribute.

3.2 Collections

The mechanism introduced in this section is used throughout TACTS to provide objects with a hierarchical structure. The idea is to define a type for objects which represent collections of other objects. Figure 3.1 depicts the part of the TACTS type hierarchy introduced in this section. The boxed type ***free** has already been introduced.

3.2.1 Object Collections

The type ***collection** describes objects which represent a family of named sets of other objects (which may be collections themselves). The sets are implemented

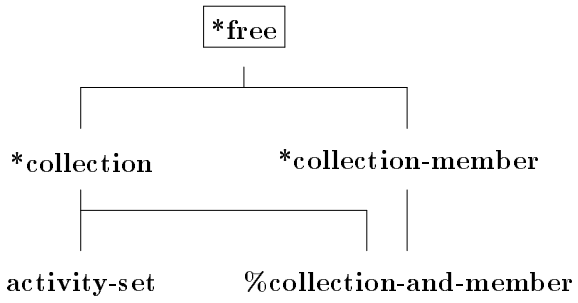

Figure 3.1: The Types for Collections

by using ELIOOS groups. The members of a set are exactly the objects in the corresponding group. This implies that an object may be a member of at most one set of at most one collection object. The main attribute of a collection object is the list of set names defined for it together with the corresponding group numbers.

This approach allows a thematical grouping of subobjects by using different sets. As an example, an object representing a structured document with history protocol is implemented by a collection with two object sets: the set of document parts and the set of activities performed on the whole document.

3.2.2 Displaying and Member Lists for User Interaction

The type ***collection** also implements the central part of the TACTS user interface. It displays lists of set members in the object buffer. Objects can be selected from this list by moving the cursor (point) to it. They can then be used in a subsequent operation.

The method **display-subs** is defined to display a list of members of one set associated with a collection object. The method uses the method **display** for every single set member to display its information. Thus the information listed for a member may depend on the member's type.

Figure 3.2 depicts the user interface. It shows a situation with two windows, each containing a list of members. The upper window displays the subactivities of the root object in the base. The lower window displays the subactivities of the activity 'example4' which is a member of the list in the upper window. Every object is displayed in a block of consecutive lines. An object may be selected from the list by moving point to it or by clicking the mouse on it. There are several operations (implemented by methods of type ***collection**) which use the selected object in some way. One example is the method **delete-sub** which deletes the selected object.

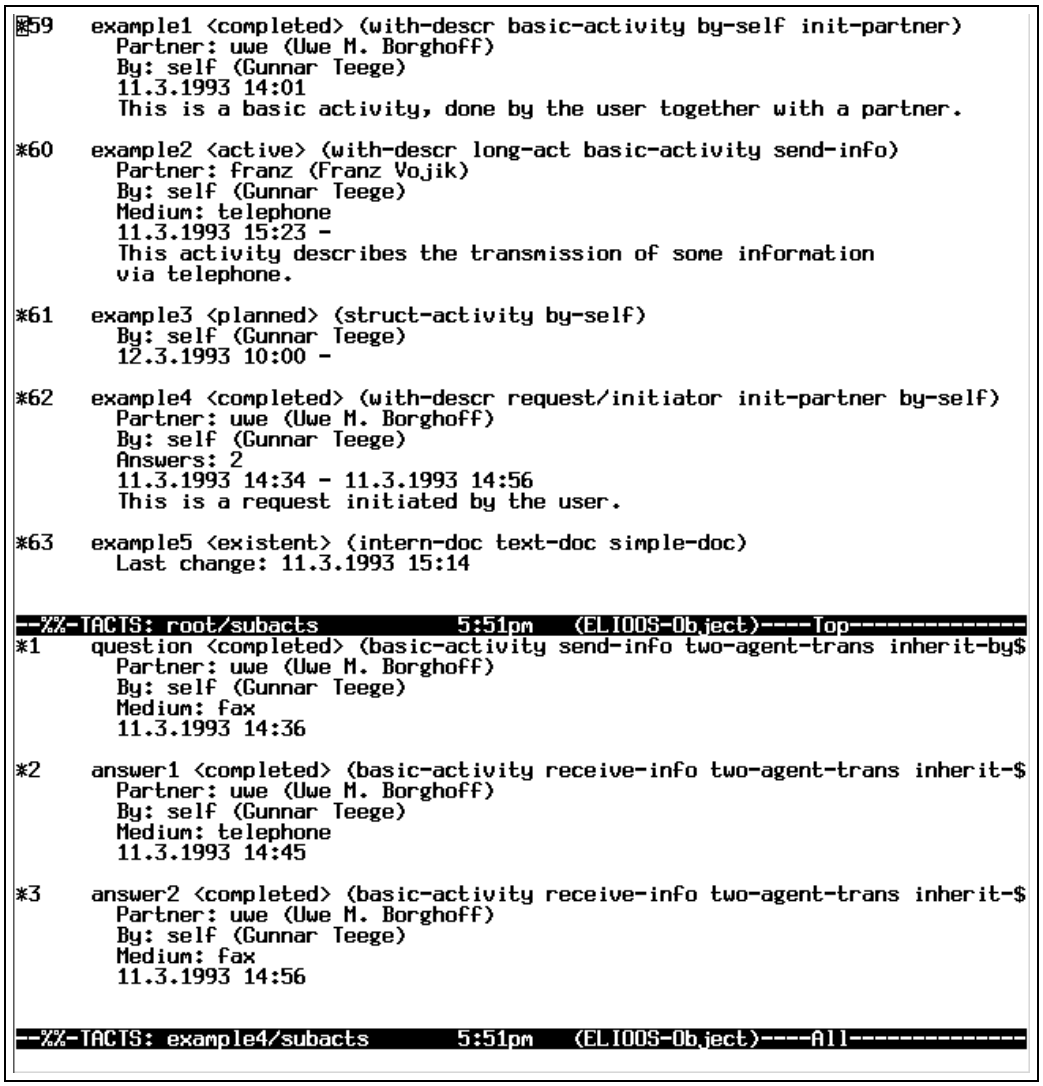
*59 example1 <completed> (with-descr basic-activity by-self init-partner)
Partner: uwe (Uwe M. Borghoff)
By: self (Gunnar Teege)
11.3.1993 14:01
This is a basic activity, done by the user together with a partner.
*60 example2 <active> (with-descr long-act basic-activity send-info)
Partner: franz (Franz Vojik)
By: self (Gunnar Teege)
Medium: telephone
11.3.1993 15:23 -
This activity describes the transmission of some information
via telephone.
*61 example3 <planned> (struct-activity by-self)
By: self (Gunnar Teege)
12.3.1993 10:00 -
*62 example4 <completed> (with-descr request/initiator init-partner by-self)
Partner: uwe (Uwe M. Borghoff)
By: self (Gunnar Teege)
Answers: 2
11.3.1993 14:34 - 11.3.1993 14:56
This is a request initiated by the user.
*63 example5 <existent> (intern-doc text-doc simple-doc)
Last change: 11.3.1993 15:14
--%%-TACTS: root/subacts 5:51pm (ELIOOS-Object)----Top----------
*1 question <completed> (basic-activity send-info two-agent-trans inherit-by$
Partner: uwe (Uwe M. Borghoff)
By: self (Gunnar Teege)
Medium: fax
11.3.1993 14:36
*2 answer1 <completed> (basic-activity receive-info two-agent-trans inherit-$
Partner: uwe (Uwe M. Borghoff)
By: self (Gunnar Teege)
Medium: telephone
11.3.1993 14:45
*3 answer2 <completed> (basic-activity receive-info two-agent-trans inherit-$
Partner: uwe (Uwe M. Borghoff)
By: self (Gunnar Teege)
Medium: fax
11.3.1993 14:56
--%%-TACTS: example4/subacts 5:51pm (ELIOOS-Object)----All----------

Figure 3.2: The TACTS User Interface

3.2.3 Other Operations on Collections

There are some primitive operations for accessing collection members. It is possible to add a new named empty set to a collection by specifying the set's name. It is possible to query the list of set names defined for a collection and, given the name of a defined set, to query the list of objects in that set. It is possible to create a new object and immediately add it to a defined set. It is possible to delete the selected object (the object ceases to exist) or to specialize its type. It is also possible to move an object from one set to another set within the same or another collection. Finally, it is possible to invoke a method for the selected object.

On the other hand, the type ***collection-member** describes all objects that are a member of a collection. It defines an operation which queries the collection and the set name the object is a member of. To ensure that all collection members are of this type[1], every object is automatically specialized to this type (if necessary) upon addition to a collection's set.

The collection objects are used to impose a tree structure upon the objects in a base. There are two simple operations for navigation in this tree. The method **go-down** of type ***collection** switches to a buffer containing the list of a set defined for the selected object (provided the selected object is itself a collection). The method **go-up** of type ***collection-member** switches to a buffer containing a list of a set defined for the collection object the current object is a member of (provided the current object is a member of some collection's set).

Finally, the type ***collection** defines the method **search-member**. It takes as arguments a set name and a predicate in the form of a lisp function. It tests all members of the named set and returns the first one which matches the predicate. The combination supplement type **%collection-and-member** is a specialization of the types ***collection** and ***collection-member**. It describes collection objects which are themselves a member of another collection. The type enhances the method **search-member** by optionally invoking the method at the superordinate collection if no member has been found locally. This implements a search which proceeds upwards until the base's root object is reached or a member has been found.

3.2.4 Activity sets

The main application of collections in the TACTS system is for activities which are divided into subactivities. All these activities are collections with a set named "subacts" containing the direct subactivities. Activities with subactivities are described by the type **activity-set** which is a specialization of ***collection**.

[1]Remember that the type of an object normally is the combination of many other types. We use the term "is of type X" to denote that X is one of the types in the combination, or a supertype of one or more of the combined types.

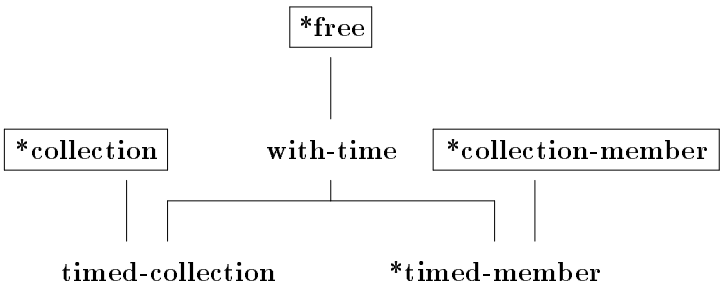

Figure 3.3: The Types for Time Specifications

Every TACTS base is structured in the form of an activity tree. The root of the tree is the activity subsuming all other activities in the base. Whenever an activity set is attached to an object buffer, it displays the list of its members in the object buffer, thus giving the user interactive access to these activities. With the help of the navigation commands described in Section 3.2.3 the user can move up and down in this tree of activities. When a TACTS base is loaded into the system, an object buffer is created and the root of the activity tree becomes attached to it, thus displaying the list of its immediate subactivities.

3.3 Time Specifications

The next mechanism is used to add a time specification to objects representing an activity. It is used to describe the time when the activity is planned to take place (if it is in the future) or actually has taken place (if it is in the past). Since activities can be large and may have an internal structure of subactivities, these time specifications must be flexible enough to allow arbitrary sets of time intervals.

Figure 3.3 depicts the part of the TACTS type hierarchy introduced in this section. The boxed types have already been introduced.

3.3.1 Interval Sets and Operations

Time specifications are implemented as finite or infinite sets of time intervals. This mechanism was developed in [9].

Time intervals are specified on a per minute basis. Every interval has a start minute and an end minute. Special cases are open time intervals and time points. Open time intervals either have a start infinitely in the past, or an end infinitely in the future. The interval with the start infinitely in the past and the end infinitely in the future is provided, too. Time points are intervals for which start and end coincide.

Time intervals are collected in sets of non-adjacent and non-overlapping intervals. A set may contain an infinite number of intervals if they can be specified by a periodic repetition of a prototype interval. A set may contain several infinite sequences generated from various prototypes of different periods. Infinite sets are a useful means of describing periodic activities such as weekly two hour meetings.

The mechanism includes naming of time specifications and functions to generate time specifications. Both are accessible to and can be configured by the user. For example it is possible to define the name "monday" as the periodic set containing all intervals from 0:00 to 24:00 on mondays and to define a function named "december" which returns the interval of the month december for the given year (note that the set of all december intervals is not periodic and thus can not be described by a single time specification).

The user interface for time specifications consists of a parser for user input and a converter for user output. The converter displays time specifications in a readable standard format. This is a list of single intervals or periodic sequences from one prototype. A single interval is given by its start and end minute displayed as date (day, month, year) and time (hour, minute). For a periodic sequence the prototype and the period are given. There are several enhancements possible such as name display if a name is defined, or displaying the date only once if it is common to both the starting and ending point. This will be done in coming versions. Currently only the abbreviation of time points as single minute specifications has been implemented.

The parser allows a rich variety of input forms. Aside from specifying an interval set explicitly or by name, specification relative to the current time is possible. Every defined name for an interval set or a generation function may be preceded by the modifiers "next", "this", or "last" and optionally by a number. In these cases the appropriate single interval is selected. Thus, if suitable definitions for the names are provided, specifications such as "this month", "next friday", "last easter", "next holiday" may be entered. The form "next 2 hour" means the second-next full hour interval relative to the current time point.

Even more flexibility is added by operations defined on interval sets. It is possible to unite, cut, invert, or subtract time specifications.

- Uniting two interval sets produces a new interval set which contains all time points described by at least one of the original sets. An example is the uniting of "monday" with "friday" to produce "monday and (or) friday". Another example is to unite "afternoon" with "weekend".

- Cutting two interval sets produces a set containing those time points described by both original sets. An example is the cutting of "monday" with "afternoon" to produce "monday afternoon". Another typical example is to cut a periodic interval like "monday" with a single larger interval such as "summer semester 1992" which results in all mondays in the summer semester 1992.

- Inverting an interval set produces a new interval set which contains all time points not described by the original set. Examples are the inversions of "monday" ("all days except monday"), of "this day" ("not today"), or of "workdays" ("holidays").

  Subtracting an interval set B from an interval set A can be done by cutting A with the inversion of B. In this way specifications like "next week except wednesday" or "holidays except easter" can be constructed.

All of these operations are fully implemented and may be arbitrarily combined. The operations include a canonization of the results. The canonization e.g., eliminates unnecessary intervals and unites overlapping or adjacent intervals. This simplification is necessary for easy readability of the output produced by the converter.

3.3.2 Operations on Timed Objects

The type **with-time** describes objects with an attribute **Time** which may have a time specification as value. The type defines three methods, namely **cuttime**, **addtime** and **subtracttime**, which apply the three binary operations to the attribute value. The second argument of the operation is read in interactively by the parser (when the method is activated by the user) or may be given as method argument (when the method is called from inside another method).

The type only handles the storing and manipulating of the time attribute. It does not restrict its semantics to the intended use of "time when the activity happens". The type may be used for objects representing other things apart from activities as well, providing them with a time interval set.

3.3.3 Combining Time Specification and Collections

When the types ***collection** and **with-time** are combined the resulting type describes collections with time specifications. For this type it is interesting to define a specialization which couples these features in certain ways. This specialization is called **timed-collection**.

The basic idea of the coupling is to inherit time specifications to and from collection members in a way that the union of all time specifications for members is contained in the time specification for the collection. This is consistent with the semantics of activity objects. Since subactivities are parts of the main activity they take place during the time in which the main activity takes place.

The coupling is implemented by extending the methods **cuttime** and **subtracttime** for the collection and by extending the method **addtime** for every member with a time specification. For the latter, a type ***timed-member** is defined as specialization of **with-time**. The methods for creating objects in a timed collection and for moving objects into a timed collection are extended to

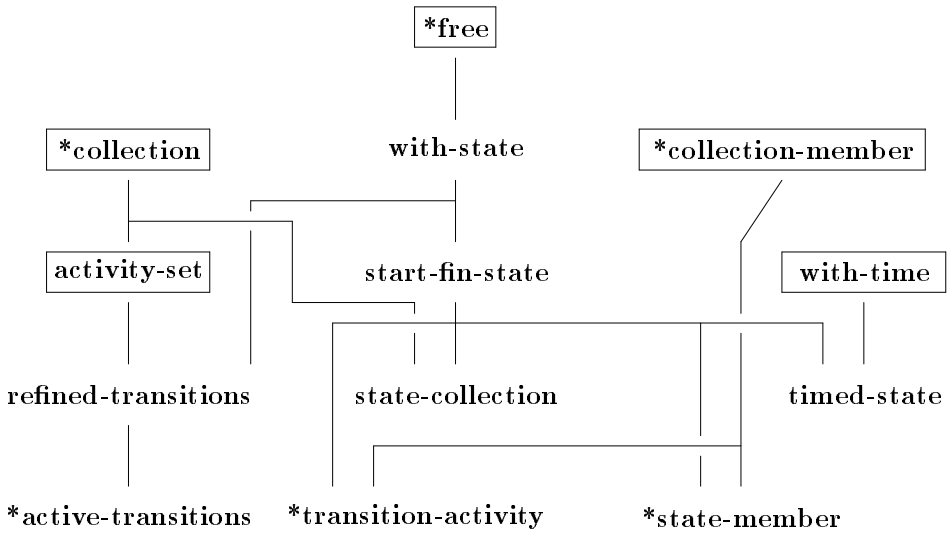

Figure 3.4: The Types for State Management

specialize these objects to type ***timed-member**, if necessary. This guarantees that all members of a timed collection which have a time specification are of this type.

The extended versions of **cuttime** and **subtracttime** simply propagate the operation to all members with time specification in all defined sets. The extended version of **addtime** propagates the operation to the collection object (if existent and if it has a time specification).

Note, that the type **timed-collection** is not a combination enhancement which is added automatically, but a user concept. Thus the user may decide whether a collection instance manages time constraints for its members or not.

3.4 State Management

The last basic mechanism is used to specify a state for activities or other objects. The possible states and the transitions between them form a finite state automaton. It enables the specification and use of execution patterns with sequential and parallel subactivities. This includes conversation patterns as in Coordinator, as well as office procedures as in the DOMINO system. The state mechanism was partly developed in [9].

Figure 3.4 depicts the part of the TACTS type hierarchy introduced in this section. The boxed types have already been introduced in the previous sections.

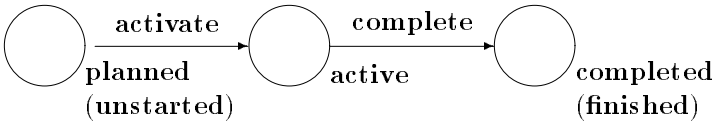

Figure 3.5: A Simple Automaton

3.4.1 Automata as Type Constituents

In TACTS, an automaton is specified by a number of *states* and a number of labeled directed *transitions* leading from one state to another. The transition labels are locally unique in the sense that in every state for every label at most one transition starts with this label. However, there may be different transitions with the same label if their starting states are different. Every state may have a sort. For the use of sorts see Section 3.4.3 below. Figure 3.5 depicts an example automaton with three states and two transitions.

Automata of this kind describe all activities which are executed according to the same pattern. Thus an automaton is not a property of a single object. Instead, like methods, automata are part of object type descriptions. The specification of an automaton consists of a sequence of commands which add or change its constituents. Among these are commands for adding a new state or adding a new transition.

Types which include an automaton may be combined like other types. During this step the automata of the original types are combined into one automaton of the resulting type. The combination is done by concatenating all automaton building commands. If only commands are used which add elements then the resulting automaton consists of the union of the states and transitions of the original automata.

This design combines the specification of execution plans with object-oriented programming and multiple inheritance in an elegant and extremely flexible way. It enables the building of complex execution plans by combining and enhancing simple basic plans, resulting in a toolkit for execution plans.

The type **with-state** describes all objects with a state that may change according to an automaton. It defines the methods necessary for the mechanism, however it only contains the empty automaton (no states or transitions defined). In this way it is a generic type and allows the incremental definition of any automaton by specialization.

3.4.2 Operations on Objects with State

Whereas an automaton is part of a type description, the actual state of an activity is object specific and stored in an attribute **State** defined by type **with-state** for any of its objects. Because we are interested in modeling activities, states

are of no great interest apart from their identity and possibly a name for output. The really important parts are the transitions. They describe those situations in which something "happens". Thus they are candidates for being structured and detailed.

Every transition consists of at least two steps. The first step leaves the source state and begins the transition. The second step enters the target state and completes the transition. The transition execution may be stopped in between these steps. Then the object is in a special state and there are only two possibilities for going on: either completing the transition, or resetting the transition by reentering the source state.

The type **with-state** defines the three methods **begin-transit**, **complete-transit**, and **reset-transit** for the single parts of transition execution. Additionally, the methods **transit-step** and **transit-run** implement the execution of transitions as a whole. The **transit-step** tries to execute one transition by beginning and completing it. If there is already a begun transition, it is completed. If no transition is specified and there is more than one possible transition in the current state, the user is asked to select one of the possibilities. The method **transit-run** performs transitions until the user stops it or it reaches a state where no transition leads away. The user is asked to select a transition or stop the execution of **transit-run** before each transition beginning.

Both methods **transit-step** and **transit-run** invoke another method **action-transit** immediately after beginning a transition. This method is defined by type **with-state**, however it does nothing. It may be used in specializations of type **with-state** to add actions to be performed by TACTS during the transition execution. The type **with-state** only implements the state change during transitions.

Thus, by simply enriching the automaton of type **with-state** and using the methods **transit-step** and **transit-run** as they are, it is possible to construct a wide variety of concrete execution plans together with an interface which keeps track of the current state and allows the state specific selection of transitions.

3.4.3 Starting and Finishing Objects

In many cases it is useful to have a notion of "starting" and "finishing" an object. This could be implemented by using an automaton with corresponding transitions. However, in TACTS a more general approach is taken. Any state in an automaton may be marked **unstarted** or **finished** by setting its state *sort*. As an example, in Figure 3.5 state sorts are given in parentheses. The state **active** does not have any specific sort.

This mechanism of state sorts allows more than one state to have these properties and more than one transition to be a "starting" or "finishing" transition. An example are two states for successful and unsuccessful completion of an activity. In both states the activity is finished.

Based on the two state sorts **unstarted** and **finished**, four kinds of transitions are defined. A transition which leads from an unstarted state to a state of any other sort is a "starting" transition. A transition which leads from any state that is not of sort **finished** to a **finished** state is a "finishing" transition. In Figure 3.5, the transition **activate** is a starting transition and the transition **complete** is a finishing transition. Any transition leading from state **planned** to state **completed** would be both starting and finishing.

Analogously, an "unfinishing" transition leads from a **finished** state to a state of some other sort and an "unstarting" transition leads from a state of some other sort to an **unstarted** state. The latter two cases are useful for activities which may be reset, restarted, or undone.

The type **start-fin-state** describes objects which may be started and finished. It is a specialization of **with-state**. Its automaton is still empty, however, the four methods **start**, **finish**, **unstart**, and **unfinish** are defined. The methods **begin-transit**, **reset-transit**, and **complete-transit** are enhanced to detect the kind of the transition based on the sorts of source and target states and, if appropriate, to invoke the corresponding new method. The new methods do not contain any specific operations. However, they may be enhanced in specializations to execute certain steps during transitions of the corresponding kind.

Note that a transition is both starting and finishing if it leads from an **unstarted** state directly to a **finished** state. In this case first the method **start** is called by the method **begin-transit** and the method **finish** is called by the method **complete-transit**. Analogously, a transition from a **finished** state to an **unstarted** state is both unfinishing and unstarting.

3.4.4 Combining State and Collections

When collections are combined with the state mechanism, it is possible to support consistency conditions for the state of all objects in a collection tree like it was shown for the time specification in the types **timed-collection** and ***timed-member** in Section 3.3.3. For states, the consistency conditions are based on the mechanism for starting and finishing objects as described in Section 3.4.3. They may be formulated as the single condition:

> Whenever a collection is in an **unstarted** or **finished** state, all members which are of type **start-fin-state** must be in a state of the same sort.

For an object of type **activity-set** this means that if it is in an **unstarted** state, all subactivities must be in an **unstarted** state as well. If it is in a **finished** state, all subactivities must be in a **finished** state as well. This condition clearly makes sense for the usual semantics of activities and subactivities.

The conditions are implemented by the type **state-collection** which is a specialization of ***collection** and **start-fin-state**, and the type ***state-member**

which is a specialization of ***collection-member** and **start-fin-state**. The type **state-collection** enhances the methods **unstart** and **finish** by testing all members for being in a state of the necessary sort. If this is not the case, the user is informed and the transition is not completed (it remains begun). The type ***state-member** enhances the methods **start** and **unfinish** by testing the collection for being in a state where this kind of transition is allowed. If it is not allowed, again the user is informed and the transition is not begun. This behavior implies that the user has to start an activity as a whole before he/she may start any subactivity of the activity, and the user has to finish all subactivities before finishing the activity as a whole.

An alternative approach would automatically start or unfinish the collection when it is necessary, and it would automatically finish or unstart all members when it is necessary. However, in general there may be more than one corresponding transition. Without knowing the automata used by the objects it is not possible to decide which transition should be activated in this case. For certain activities with a fixed automaton which contains transitions of all four kinds in every state, this behavior could be defined as specialization replacing the approach described in this section.

3.4.5 Combining State and Time Specification

The type **timed-state** couples time specifications and automata. The main idea is to automatically set the time during certain transitions. This is again based on the mechanism of starting and finishing objects as described in Section 3.4.3.

Whenever a starting transition is begun, the start time is set to the current time by cutting the activity's time specification with the interval from the current minute to infinity. In the same way the end time is set whenever a finishing transition is completed. If a transition is both starting and finishing, both times are set, and the result is a time point specification.

For every unfinishing transition, the time specification is united with the interval from the current minute to infinity. This models restarting of an activity by adding a new time interval. An activity which was restarted several times thus has as time specification the set of exactly those time intervals in which it was not in an **unstarted** or **finished** state.

For every unstarting transition, the time specification is cut with the interval from the current minute to infinity. This models undoing the activity by forgetting about the existing time specification and initializing it anew, i.e., to the interval it may take place from now on.

The type **timed-state** is implemented as a specialization of **start-fin-state** and **with-time**. The time setting steps are added to the respective methods **start**, **finish**, **unfinish**, and **unstart**. Of course, the types **timed-state**, **state-collection**, and **timed-collection** may be combined resulting in a type which couples all three basic mechanisms.

3.4.6 Executing a Transition upon Instance Creation

The automaton of Figure 3.5 may be used to model the execution of activities. Upon creation, all these activities are in state **planned** and must explicitly be activated.

In many cases it is convenient to enter unplanned activities which should be activated or completed immediately after creation. This is possible with the help of type **immediate**. It specializes the generator method **created** by executing the method **transit-step** after initializing the state.

This works for all objects using the state mechanism. If there is only one transition leaving the initial state, such as in Figure 3.5, then it is executed. If there are several transitions, the user is asked to select a transition, as usual in the method **transit-step**.

3.4.7 Refined Transitions

The approach of Section 3.4.2 is further extended by allowing hierarchically structured dynamic automata. The key idea to do this is the explicit modeling of transitions by one or more activity objects which may have their own automaton. These activities are subactivities of the main activity, thus we chose as starting point a combination of types **activity-set** and **with-state**. The behavior of the overall activity is specified by the automaton of the main activity object together with all automata of the subactivity objects.

We call transitions with associated subactivities *refined* transitions and the subactivities *transition activities*. Every refined transition may have one or more transition activities. A transition is completed when *all* its transition activities are finished. Refined transitions remain begun until the last transition activity is finished. Then it is automatically completed. If all transition activities are reset to an unstarted state, the transition is automatically reset to its source state.

The activities associated with a transition may be executed in any order including parallel. At any time it is possible to switch from one to another. Furthermore, activities may be created and added to a transition even when the transition is begun. This allows the use of transitions with a previously unknown number of concurrent subactivities.

To facilitate such refined transitions, it is necessary to determine when an associated activity is finished. Here, the mechanism described in Section 3.4.3 is used again. Thus, every transition activity must be of type **start-fin-state**. The type ***transition-activity** defines the behavior of transition activities. It is added automatically to every object used as a transition activity.

The type which implements refined transitions is called **refined-transitions**. As described above, it is a specialization of **with-state** and **activity-set**. The methods **transit-step** and **transit-run** are enhanced as follows. Immediately after calling the method **action-transit** they try to execute one or more transitions

for a transition activity. The method **transit-step** calls itself recursively, thus only one transition is executed at a transition activity. The method **transit-run** also calls itself recursively, thus transitions for transition activities are performed until the user stops it or there are no more transitions applicable. Then the method offers other transition activities, or, if all transition activities are finished, it completes the main transition and offers to begin the next one.

In both cases, first a transition activity must be selected. If there is only one unfinished transition activity, the method **transit-step** selects it without asking the user. In all other cases the user may select among all unfinished transition activities. In the method **transit-run** the user may additionally stop the execution.

Additionally the type **refined-transitions** defines the new method **add-trans-act**. It may be used to explicitly associate an (existing) subactivity with a transition.

3.4.8 Programming of Execution Plans with Refined Transitions

TACTS provides a simple mechanism for the user to specify automata with refined transitions. The user may write type definitions which specify the automaton and the transition activities. The automaton is given by introducing the states and the transitions with their source and target states. For each transition a fixed number of transition activities may be specified. Each transition activity is specified by a name, a type list, and a list of parameter bindings for the method **created**. The name is used as a default for the shortname of the instance and to identify the transition activity in type specializations.

The instances of a type defined in this way behave as follows. During the begin of a transition an instance of every transition activity is created. After completing the transition, the transition activities are no longer associated with the transition. Thus, if a transition may be executed more than once (because it lies on a cycle in the automaton), every execution uses its own set of transition activity instances.

This behavior is defined by the type ***active-transitions**, which is a specialization of the type **refined-transitions**. The user writes his/her types as specializations of ***active-transitions**. Optionally the type **start-fin-state** may be added and the types introduced in section 4.1.1 may be added to specify (a part of) the automaton.

The user may also define specializations of his/her own types. In a specialization it is possible to extend the automaton by adding states and/or transitions, to specify additional transition activities for transitions, or to add types or parameter bindings to inherited transition activities. In the last case the transition activity's name is used to identify the transition activity which has to be changed.

On the other hand, the type of a transition activity may be a specialization of type ***active-transition** too. It may even be identical to the main type. This results in hierarchical automata of depth greater than two.

The type ***active-transitions** provides a powerful mechanism to specify hierarchical execution plans with a static scheme. However, the user may always leave the scheme and create additional subactivities manually, thus unexpected cases may be handled easily. For specifying the execution plan the user has to write type definitions in a lisp-like syntax. This will be improved in future versions of TACTS by providing a specific language for defining execution plans.

Chapter 4

The Core of TACTS

This chapter describes the core of the TACTS system. It defines several basic types of activities. Only history recording is provided for these activities in the core, other support for the execution of the activities is added in extensions.

Currently the core part is rather simple, it only supports activities which involve one or two users. The extension of the core will be an important task in future versions of TACTS.

4.1 Activities

The most important kind of objects used in the TACTS system are activities. The TACTS core defines types for basic activities and activities with subactivities. The Figure 4.1 depicts the corresponding part of the type hierarchy. As usual, boxed types were introduced in earlier sections.

4.1.1 The Basic Automata

The activities defined in the TACTS core are characterized by their state automaton. The current version of TACTS provides some simple automata that apply to a broad range of activities. All of them are extensions of a basic two-state automaton defined by the type **simple-act** and all of them use the starting/finishing mechanism introduced in Section 3.4.3. Just the automaton itself is extended, the methods are left unchanged. For every feature a separate user concept type is defined. They may all be combined in arbitrary ways.

The type **simple-act** is a specialization of type **start-fin-state**. The automaton consists of one **unstarted** state **planned**, one **finished** state **completed** and one transition **complete** from state **planned** to state **completed**. The **simple-act** automaton describes short activities which are executed at a single time point. Activities of this kind may not be undone or canceled.

Activities which last some time are described by the specialization **long-act**

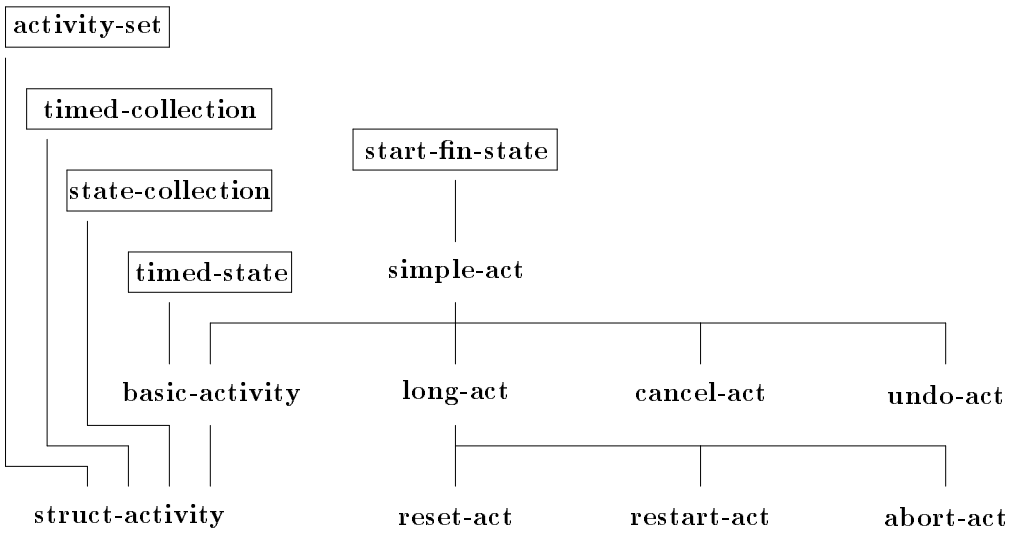

Figure 4.1: The Basic Types for Activities

of type **simple-act**. It adds a new unsorted state **active** and two transitions **activate** from state **planned** to **active** and **complete** from state **active** to **completed**. Additionally, it removes the transition **complete**, leading from state **planned** to state **completed**. The resulting automaton is depicted in Figure 3.5.

Activities which may be canceled are described by the specialization type **cancel-act** of type **simple-act**. It adds the state **canceled** of sort **finished** and the transition **cancel** from state **planned** to **canceled**. Activities which may be undone are described by type **undo-act** which adds the transition **undo** from state **completed** to **planned**.

Specializations of type **long-act** are the type **reset-act** with an additional transition **reset** from state **active** to **planned**, the type **restart-act** with an additional transition **restart** from state **completed** to **active**, and the type **abort-act** with an additional state **aborted** of sort **finished** and a transition **abort** from state **active** to **aborted**.

The combination of all these types results in the five-state automaton depicted in Figure 4.2. Future versions of TACTS will further extend this toolkit of execution plan features.

4.1.2 Basic Activities and Structured Activities

The type **basic-activity** is a user packet type, combining the types **timed-state** and **simple-act**. It may be used to represent unstructured activities which are performed in a single step. For each instance of type **basic-activity** the user may add a description by adding the type **with-descr** (see Section 3.1.2), or

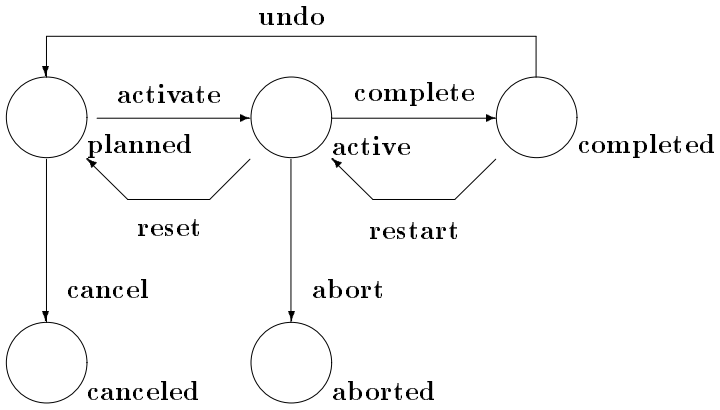

Figure 4.2: Combining All Predefined Execution Plan Features

he/she may add other execution plan features by adding the corresponding types described in Section 4.1.1.

The type **struct-activity** is a user packet for structured activities. It combines the types **basic-activity**, **long-act**, **timed-collection**, **state-collection**, and **activity-set**. Thus, all structured activities automatically use the three-state automaton depicted in Figure 3.5. They maintain consistency conditions between their own state and time specification and those of their subactivities.

4.2 Agents

Besides activities, TACTS also uses *agents*. An agent may represent a person, a group of persons, an institution, or any other instance that may be involved in an activity.

4.2.1 Agent Objects

The TACTS core implements objects which represent agents by the type **agent**. In its simplest form, an agent is only described by its shortname (like any TACTS object). The shortname can be used like an alias. It is used for interactively specifying an agent and for displaying agent references in other objects.

Additional informations about agents are implemented by specializations of the type **agent**. Examples for additional information are a name, an address, a telephone number, or an electronic mail address. The applicable information depends on the kind of the agent. Thus, every information item should be defined by a separate type. Then for any kind of agent, an individual type combination may be specified. However, for accessing this information in a uniform way, the type **agent** defines the method **agent-info**. It is called with a keyword and returns corresponding information, if defined. No keywords are predefined by

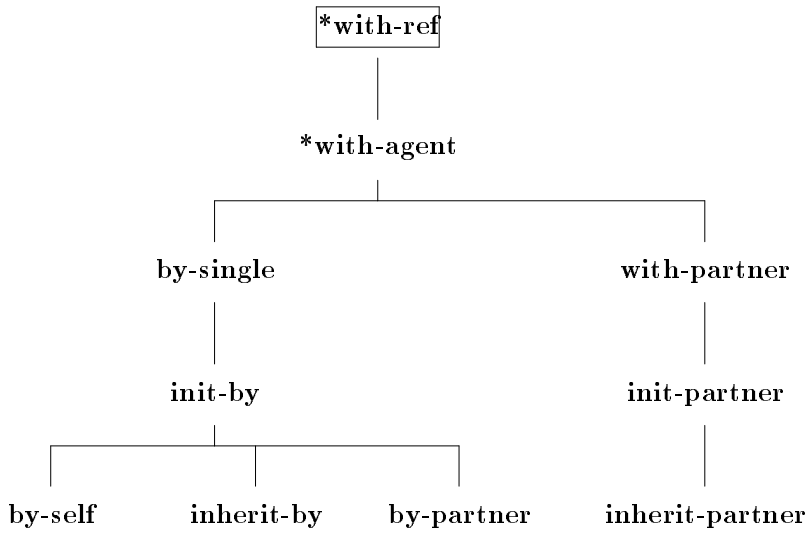

Figure 4.3: Activities with Agents

type **agent**, all keywords must be defined and implemented by specializations. See Section 7.1.1 for an example.

4.2.2 Agent Sets

Agents are represented by objects and thus belong to a base. The type **agent-set** describes collection objects with an associated set, named "agents". Any object may be an agent set. If the object represents an activity, the agent set should contain all agents which are involved in this activity only. The root object of the object tree in a TACTS base always is of type **agent-set** and the associated set contains all agents which are not local to a specific activity.

The method **show-agents** of type **agent-set** pops up a buffer with a list of the agent set members. Using this buffer, the user can create any number of agents for later use in activities. Like other objects, agents can be moved and deleted.

4.2.3 Activities with Agents

Agents may be associated with activities for several purposes. Examples are the executing agent, the initiating agent, or agents that are involved in the activity in other ways. Every kind of agent is associated with the activity by storing a reference to the agent object in a corresponding attribute. Figure 4.3 depicts the types for activities with two kinds of agents, defined in Sections 4.3 and 4.4.

The type ***with-agent** is a specialization of type ***with-ref**. It defines or enhances methods to manipulate agent references. It does not define concrete

agents. Thus it is not a user concept. All methods take as argument the name of the attribute containing the reference.

The method **set-agent** is used to set the reference for a given kind of agent. If no reference is given, there are two possibilities for setting the agent interactively. First, the method displays a menu of all agents defined at the collection the object is a member of. The user may select one of them, thus copying the agent reference from the superordinate object to the subordinate object. If no agent is selected, the method asks the user for the shortname of the agent. The method then looks up the agent in all agent sets on the path from the current activity to the root object of the current base with the help of method **search-member** (see Section 3.2.3). The first agent found with the given shortname is used and a reference to it is stored using the inherited method **set-ref**.

When an object with associated agents is displayed, the agent is displayed by its shortname. If no reference is stored in the attribute, the string "<not given>" is displayed instead. If the reference can not be resolved, it is assumed that the agent object was deleted and the string "<deleted>" is displayed.

4.3 The Executing Agent of an Activity

The simplest form of activities supported by the system are activities executed by a single agent. This may be the user of the system or any other person. In the latter case the TACTS user only protocols the activity.

4.3.1 Activities Executed by a Single Agent

The type **by-single** defines a new attribute **By** that stores an agent reference. Semantically, it should be the agent that executes the activity. As usual, the type **by-single** may be combined with other types like **basic-activity** and **struct-activity** and specializations thereof.

The type **init-by** is a specialization of type **by-single**. It sets the executing agent upon creation of the object, using the method **set-agent**.

The type **inherit-by** further specializes type **init-by**. It does not ask interactively, but tries to use the executing agent of the containing collection.

4.3.2 Activities Executed by the User

A special form of activities with a single executing agent are those activities executed by the TACTS user himself. The type **by-self** describes activities of this kind. It is a specialization of **init-by** which automatically tries to set the **By** attribute to a reference to an agent with shortname "self".

To use this type, the corresponding agent object must exist. TACTS automatically creates an agent with shortname "self" in the agent set of the root

object in each base.

4.4 The Partner Agent of an Activity

Additionally to the executing agent, TACTS supports the specification of a partner agent. No further assumption is made in the TACTS core about the way the partner is involved in the activity.

4.4.1 Specifying an Activity Partner

The partner specification is implemented by the type **with-partner** which adds the agent reference attribute **Partner**.

The type **init-partner** is a specialization of type **with-partner**. It sets the partner agent upon creation of the object, using the method **set-agent**.

The type **inherit-partner** further specializes type **init-partner**. It does not ask interactively, but tries to use the partner agent of the containing collection.

4.4.2 Using the Partner Specification

The partner specification may be used by optionally inheriting it to subactivities. In general, this is done by defining a specialization for all applicable types (those which specify an agent which may be the partner) using the partner specification of the containing collection as default for the agent.

An example is to use the partner of the containing collection as executing agent for an activity. This is implemented by the type **by-partner**. It is a specialization of type **init-by** that works analogous to the type **inherit-by**.

Other specializations of activities may use the partner agent in other ways. Currently, it is used by information transmission activities to specify the source or destination agent, respectively (see Section 4.5 and Chapter 7).

4.5 Basic Communication

The last part of the TACTS core describes special activities which incorporate communication. Only rather general types of communication activities are defined in the core. Figure 4.4 depicts the corresponding part of the TACTS type hierarchy. For more specific communication mechanisms see Chapter 7.

4.5.1 Information Transmission

The basic communication activity is the transmission of information via a certain communication medium. The type **trans-info** defines the user-editable string

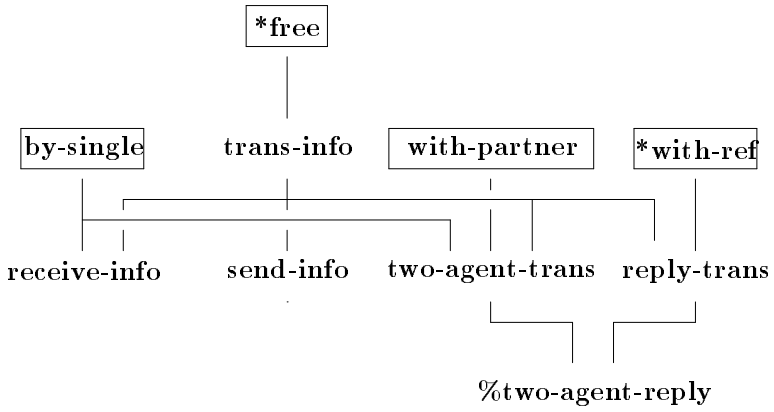

Figure 4.4: Communication Activities

attribute **Medium** and describes activities of this kind. The transmitted information itself is not supported by the type. For simple cases it may be described using the **Descr** attribute of **with-descr**.

The medium is only used for documentation purpose, thus it may be any string like "telephone", "personal talk", or "letter". However, in specializations described later, some values of the attribute **Medium** will be interpreted by the system.

There are two specialization types **send-info** and **receive-info** of the combination of the types **trans-info** and **by-single**. They describe the transmission activity from the view of the executing agent. Thus the combination of **send-info** with **by-self** describes the activity of the user of sending some information out, whereas the combination of **receive-info** and **by-self** describes the activity of the user of receiving some information.

Another simple specialization of type **trans-info** is the type **reply-trans** which describes transmissions in reply to other transmissions. To specify the other transmission, type **reply-trans** additionally inherits from type ***with-ref** and introduces the attribute **Reply-of** to store a reference to the other transmission.

4.5.2 Communication with an Activity Partner

Frequently, information is transmitted between two agents. While the type **trans-info** may also be used to describe information transmissions like broadcasts or posting to information boards, the specialization type **two-agent-trans** describes the activity of one agent of transmitting information between him/herself and a second agent. The type is a combination of the types **trans-info**, **by-single**, and **with-partner**. Here, the attribute **Partner** is used to store the second agent involved in the transmission.

The type **two-agent-trans** may be combined with the types **by-self**, **inherit-**

by, or **by-partner** to specify the executing agent, it may be combined with the type **inherit-partner** to specify the partner agent, and it may be combined with the types **send-info** or **receive-info** to specify the direction of the transmission.

If the type **two-agent-trans** is combined with the type **reply-trans**, the combination supplement type **%two-agent-reply** is added. It automatically sets the agents of the reply from the agents of the other transmission, whenever the attribute **Reply-of** is set to a reference to another transmission.

Chapter 5

Requests

This chapter describes the types for requests. They should be useful on their own, however, mainly they are an example of complex activities which are executed by a team of more than one agents. Figure 5.1 depicts the corresponding part of the TACTS type hierarchy.

5.1 Classes for Team Activities

This section describes the mechanism for defining virtual user concept types which specify activities for a whole team. As a special case the virtual type **request** is described.

5.1.1 Team Activities

All types defined in the TACTS core specify activities which are performed by a single agent. In collaborative environments, however, there is a need to specify activities performed by a team as a whole. Typically, these are structured activities and the parts are performed by separate single team members. In this way every team member plays a certain role in the team activity.

The team activity, like other activities, may be represented by an object in the user space of each team member. Currently, shared objects are not supported in TACTS / ELIOOS, thus no single object can represent the team action independently from a member. It is planned to add this feature in future versions of TACTS.

On the other hand, the member-specific object representing the team activity should also be specific to the role the member plays in the activity. Thus, every member should have an own view of the activity by maintaining an own variant of the team activity. Different views should allow different operations on the activity. This can be achieved by defining a separate type for every role of the executing agent.

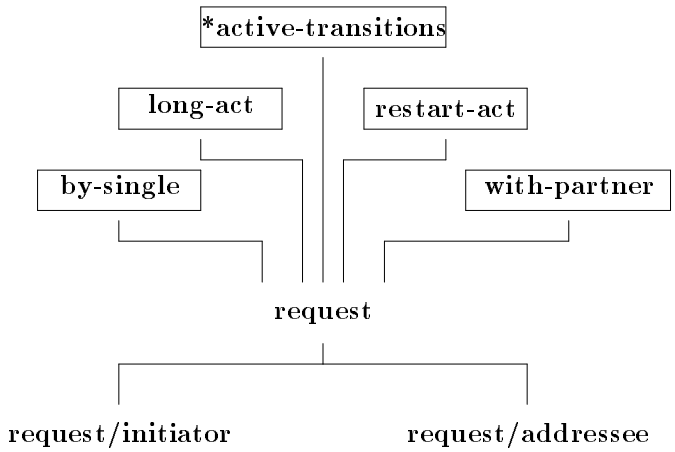

Figure 5.1: The Types for Requests

5.1.2 Virtual User Concept Types

The features of a team activity which are common to all roles should be collected in a single type. Then the roles may be specified by specializing this type in various ways.

One characteristics of the type common to all views is that it must be specialized before it can be used to create an instance of a team activity with a view. Every instance is in the user space of a team member and thus should know about the role of that member. This is the reason why we call types of this kind *virtual* types.

However, it may be convenient to create an activity object which is not yet role-specific, and later determine the role to be used. This can be done by creating an instance of a virtual type. No role specific operations can be used on this object, but at a later time a role may be specified by specializing it to a non-virtual type.

If the team activity is executed following a certain scheme, this scheme is a good candidate to be defined in the virtual type. The scheme may be defined using the state mechanism introduced in Section 3.4. In this way it is possible, e.g, to define activity patterns in the kind of those used in the Coordinator [11]. We will demonstrate this for the simple example of a request between two agents.

5.1.3 The Virtual Type for Requests

Now we describe the virtual user concept type **request**. It specifies a question of one agent (the initiator) to another agent (the addressee) together with one or more answers. The initiator and the addressee implicitly form the team which is involved in the request as a whole.

The type **request** is a specialization of the types **by-single** and **with-partner**

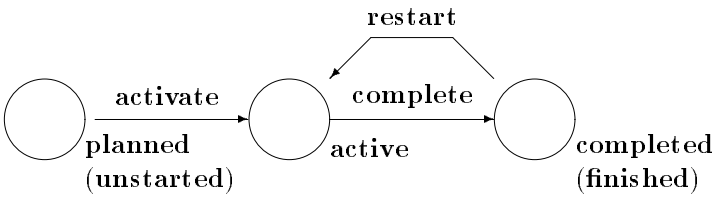

Figure 5.2: The Automaton Used for Requests

to record the two agents involved. The subactivities of every request are the transmission of the question from the initiator to the addressee and all transmissions of answers from the addressee to the initiator.

The execution scheme of type **request** is defined with the mechanism introduced in section 3.4.8. The automaton is defined by the combination of the types **long-act** and **restart-act** which results in the automaton depicted in figure 5.2. Both the transitions **activate** and **complete** each have one transition activity associated, which is of the types **basic-activity**, **two-agent-trans**, **inherit-by**, and **inherit-partner**. The transition activity of transition **complete** additionally contains the type **reply-trans**. The transition **restart** has no associated transition activity.

The two transition activities each use the automaton of type **simple-act**, like all activities of type **basic-activity**. Since the single transition of this automaton both starts and finishes the transition activity, the transitions of the request all can be done in a single step.

The transition activity of transition **activate** models the transmission of the question. Since the request can not be undone or reset, the transition **activate** may only be executed once and thus only one question transmission instance can be created. Every answer is associated to the transition **complete** in the state **active**. Since the request is of type **restart-act**, this transition can be executed more than once. Upon every execution a new answer subactivity instance is created and completed.

The type **request** could be specialized in many ways. It is possible to specialize or change the types of the subactivities. It is possible to associate more than one subactivity with every transition. It is also possible to enhance the automaton for the request and allow for more than one question.

5.2 Fixing a Role in Team Actions

To use the virtual type **request** it is necessary to fix the role of the executing agent in the request. There are two possible roles: the initiator and the addressee.

5.2.1 Role-Specific Types

Every role of a virtual type is specified by a specialization of the type. Usually, the specialization presets certain attributes, specializes the type of some subactivities, and defines additional operations which may be executed by the agent playing this role.

In the following this is demonstrated for the two roles of the type **request**.

5.2.2 The Role-Specific Types for Requests

In a request, the difference between the two roles only lies in the direction of the information transmission subactivities. When the executing agent is the initiator, the question is sent, and the answers are received. When the executing agent is the addressee, the question is received, and the answers are sent.

The initiator role of type **request** is specified by the type **request/initiator**. It adds the type **send-info** to the question transition activity and the type **receive-info** to the answer transition activity.

Typically, the instance of a request will first be generated locally by the initiator. It is in state **planned**. When the initiator executes the transition **activate** the transmission of the question is registered by the system by creating and completing the transition activity instance and the request changes to state **active**. When the first answer arrives, the initiator executes the transition **complete** and the request is completed. If an additional answer arrives, the initiator first executes **restart** and then **complete**, thus creating an additional transition activity instance of type **receive-info**.

The addressee role of type **request** is specified by type **request/addressee**. It adds the type **receive-info** to the question transition activity and the type **send-info** to the answer transition activity.

Typically, a request only becomes known to the addressee when the question arrives. Then the addressee locally creates the **request/addressee** instance. The type **immediate** may be added to execute the transition **activate** during creation, thus recording the reception of the question. When the addressee wants to send the first answer he/she executes the transition **complete**. If the addressee wants to transmit an additional answer he/she uses the transitions **restart** and **complete**.

Note that there is no connection between the initiator role instance and the addressee role instance of the request. They are in different user spaces and the system can not check or guarantee the consistency of the instances. This will be improved when shared objects enable the maintenance of a common object for the request as a whole.

On the other hand, by separating the roles it is possible that only one of the team members uses the TACTS system to maintain his/her role, while the other

member handles the request without system support informally. This greatly improves the flexibility of use of the system for single users and its wide acceptance.

Chapter 6

Documents

This chapter describes the support of activities involving documents. This part of TACTS is carefully designed to support a wide range of document and activity kinds. It supports documents stored in electronic form as well as documents held outside of the system.

The properties of documents in TACTS are discussed in separation in the following sections. The first section introduces abstract documents which have a minimum of properties. The other sections each describe types for one aspect like the document structure, the way it is stored, or the version management. For every aspect one or more types are defined, all of which are specializations of the type for abstract documents. The result again is a toolkit which supports a wide variety of document kinds by combining the types for the different aspects in several ways. Figure 6.1 depicts the types introduced in the first three sections of this chapter.

6.1 Abstract Documents

The basis of all documents is formed by the types **doc** and **edit-doc**. They define a minimum of attributes and operations common to all documents. Most of the operations are only virtual in that they do not cause any action other than optionally changing the state of the document. Thus if the types **doc** and **edit-doc** are used on their own they can only support recording the state of some document outside the TACTS system.

The types specify a fairly abstract view of documents. It is independent from features like the structure, the contents, the storing, and the use of the document, allowing for different variants for every feature and their combination by forming specializations. The main property of an abstract document is the fact that it may have some sort of content which may be altered and used. At this level any kind of document structure is part of that content.

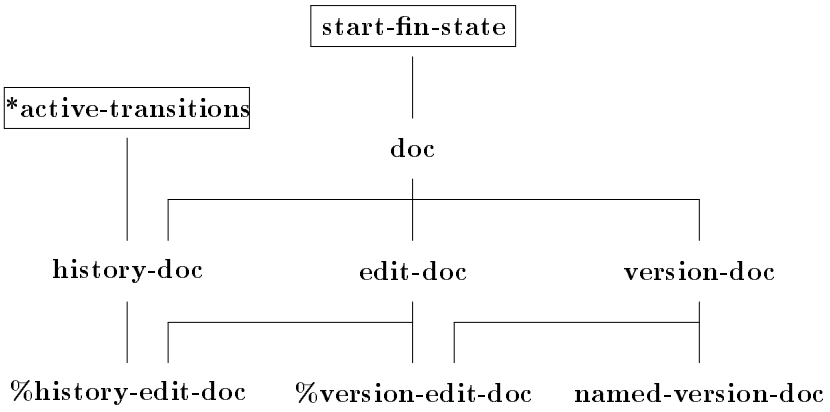

Figure 6.1: The Types for Documents with History, Versions, and Structure

6.1.1 Operations on Abstract Documents

The type **doc** defines a set of operations applicable to abstract documents (instances of type **doc**) and thus to any specialized form of documents. The type **edit-doc** is a specialization of type **doc** and defines additional operations. The operations are described together with their intended meaning in this section. Each of them is defined by a method in the corresponding type.

A document of type **doc** may be *initialized*. Initializing a document is different from creating it, because initialization applies to the content and possibly the structure of the document. Thus an existing document may be initialized more than once. The corresponding method is **init-doc**.

Analogously, a document of type **doc** may be *erased*. This only applies to the content or structure and thus differs from deleting the document instance. However, every document is automatically erased immediately before it is deleted. The corresponding method is **erase-doc**.

A document of type **doc** may be *fixed*. For a fixed document the content may not be changed any more. Thus initializing, erasing, or editing a fixed document is no longer allowed. The corresponding method is **fix-doc**.

A document of type **doc** may be *exported* to some "target". This is a very general operation which covers all kinds of document usage like previewing, printing, processing or sending out. It is possible to export the whole document or to export only the changes made to its contents / structure since the last export to the same target or some specified target. The export target is specified in two parts by giving a "system" and an "address". As an example, the "system" may be "printer" and the "address" specifies the particular printer to use. The corresponding method is **export-doc**. It has the parameters **System**, **Address**, **Changes-system**, and **Changes-address**. The last two parameters may be used to specify that only changes relative to the last export to that target shall be exported.

Dually to exporting a document, a document of type **doc** may be *imported*. This covers all cases when the content of the document is taken from a "source", like receiving it via electronic mail or from a generating system. The source is specified by "system" and "address" like the target of exporting. The corresponding method is **import-doc**.

For abstract documents one export system and one import system is predefined. The export system "document" exports the contents to another object of type **doc**. The target object is given as address of the **export-doc** operation. It works by using the import system "export" at the target object with the source object as address. This import system may not be used directly; it must rather be called by the export operation of another object. Together, both systems provide a content / structure copying feature between objects of type **doc**.

Besides these two systems, any string may be given as system and address for **export-doc** and **import-doc**. Normally, they will not be interpreted by TACTS. However, there are several specializations of **doc** which define and implement additional systems for export and import, respectively.

The type **edit-doc** supports all these operations as well. Additionally, a document of type **edit-doc** may be *edited*, which means that the content / structure is changed. Editing is done in an edit session which may last some time. Accordingly, there are two separate operations for starting and ending an edit session. The corresponding methods are **start-edit-doc** and **stop-edit-doc**.

6.1.2 The States of Abstract Documents

The most important attribute of an abstract document is its content state. It is modeled with the help of type **start-fin-state** (see Section 3.4.3). Here, a three-state automaton is used with the states **undefined**, **existent**, and **fixed**. The state **undefined** has the sort **unstarted**, the state **fixed** has the sort **finished**, and the third state has no sort. The automaton of type **edit-doc** has the additional sortless state **in-edit**.

The transitions correspond exactly to the methods for operations described in Section 6.1.1. However, not all operations may be done in every state. Some of them change the state and some leave it unchanged. Figure 6.2 depicts the automaton of type **edit-doc** with all its transitions. The transitions are not refined. However, the method **action-transit** (see Section 3.4.2) is enhanced to invoke the method for the corresponding operation. The operation methods are not designed to be called directly by the user. Instead, the user interface to the document operations is the general user interface to state transitions, i.e., the methods **transit-step** and **transit-run** of type **with-state**. The operations are executed during the corresponding transitions.

Another attribute of **doc** instances is the time of the last content change, called **Last-change**. The methods **init-doc**, **erase-doc**, **stop-edit-doc** and **import-doc** set this attribute to the current time. It is used for restricting ex-

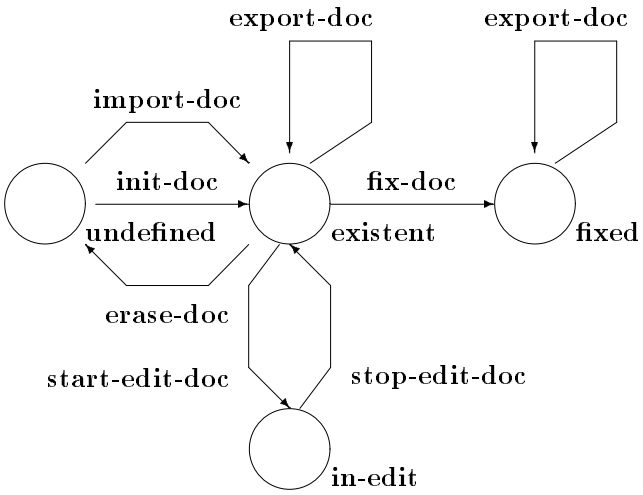

Figure 6.2: The Automaton of Type **edit-doc**

ports to changes. There is an additional attribute **Export-times** which records the time of the last export for every possible export target specifically. By comparing these time stamps the export method can determine whether the content has to be exported or not.

6.2 Document History

The history of a document is recorded by protocoling all operations applied to the document.

6.2.1 Modeling Document Operations by Subactivities

To protocol all operations applied to the document, the mechanisms from the basis and core of TACTS are used. With the help of refined transition every operation is modeled by an explicit subactivity instance. Thus, when document history is used, the object represents not only the document, but also all activities on it. It becomes an activity set with subactivities.

History for abstract documents is defined by the type **history-doc**. It is a specialization of the types **doc** and ***active-transitions** (see Section 3.4.8). If the type **history-doc** is combined with type **edit-doc**, the combination supplement type **%history-edit-doc** is added which manages protocoling of editing sessions.

6.2.2 The Subactivities for the Document Methods

In detail, the methods **init-doc**, **erase-doc**, **export-doc**, and **import-doc** are modeled by transition activities of type **basic-activity**. As usual, the transition activities are created upon beginning the transition in the automaton of type **doc**. Their transition **complete** is performed after the document method was successfully called by method **action-transit**. In particular, their time specification is set automatically to the time point when the operation was executed. A description may be added manually using the type **with-descr** (see Section 3.1.2).

For documents of type **%history-edit-doc**, every editing session is modeled by a single subactivity which has the types **basic-activity** and **long-act**. It is no transition activity. Instead, it is created and activated during the transition **start-edit-doc** and is completed during the next execution of transition **stop-edit-doc**. Thus, it records the beginning and end time of the session.

6.3 Document Versions

Version management holds all versions of the document content until the whole document object is deleted. For abstract documents versions are defined, although no content is associated with the versions. Version management may be combined with history management without problems.

6.3.1 Definition of Document Versions

In TACTS, versions are defined implicitly as follows:

> A *version interval* is defined by every maximal time interval for which the object was in one of the states **existent** or **fixed** and no state change occurred.

This definition has several implications. First, the content can not change during a version interval, thus one "version" of the content is associated with every version interval. Second, all document operations apart from **export-doc** end or start a version interval. Third, usually there are time intervals in which no version exists, i.e., the time intervals for which the document is in state **undefined** or **in-edit**. Finally, with any given time point at most one version interval is associated. Thus version intervals can be referenced by time points.

Version management for abstract documents is defined by the type **version-doc** which is a specialization of **doc**. The type uses an additional attribute **Versionhistory** to store the existing versions, that is, the corresponding time intervals as a list. All transitions which start or end a version are extended to change this attribute accordingly. The type **version-doc** may be combined with type **edit-doc** to yield editable documents with version management. The

necessary additions are defined by the combination supplement type **%version-edit-doc** which is automatically added to any combination of the types **version-doc** and **edit-doc**.

6.3.2 Operations for Document Versions

The type **version-doc** defines the new method **display-versions**. The method simply shows the version list in a separate buffer.

The single main interface to access document versions is constructed using the export operation. The method **export-doc** is enhanced to take a reference to a version interval (a time point) as additional argument. If a version interval is defined at that time point, the corresponding content is exported instead of the current content. In this case the arguments **Changes-system** and **Changes-address** of method **export-doc** are ignored. However, it would be possible to define a specialization of type **version-doc** which exports the changes between two versions in an appropriate way, e.g., by using the UNIX[1] diff command.

The version interface via the export operation is very flexible. It makes content versions usable for any export system defined for the document. In particular, if the corresponding export systems are provided, any version of the document may be previewed, printed, or processed in some other way.

For any document the version mechanism can be used in combination with the export system "document" (see Section 6.1.1). It allows to copy any version of a document to another document object. Note that the version history for a single object is linear. However, by copying versions to other objects, it is possible to construct version trees for a document. In this case every branch of the tree is represented by a separate document object with an own linear version history.

6.3.3 Named Versions

The type **version-doc** uses time points to reference versions. In other systems, like RCS, SCCS, etc., versions are usually referenced by arbitrary numbers like "1.4" or "5.23.2". TACTS provides this version naming scheme in a more general way: any version interval may be named by an arbitrary string.

This feature is implemented by the type **named-version-doc**. It is a specialization of type **version-doc**. It uses an additional attribute **Versionnames** to store a mapping from names to version intervals and defines two new methods **define-version-name** and **delete-version-name** to alter this mapping. The method **export-doc** is enhanced to accept version references by name as default. The method **display-versions** is enhanced to additionally display all defined version names in the list of version intervals.

[1] UNIX is a registered trademark of AT&T

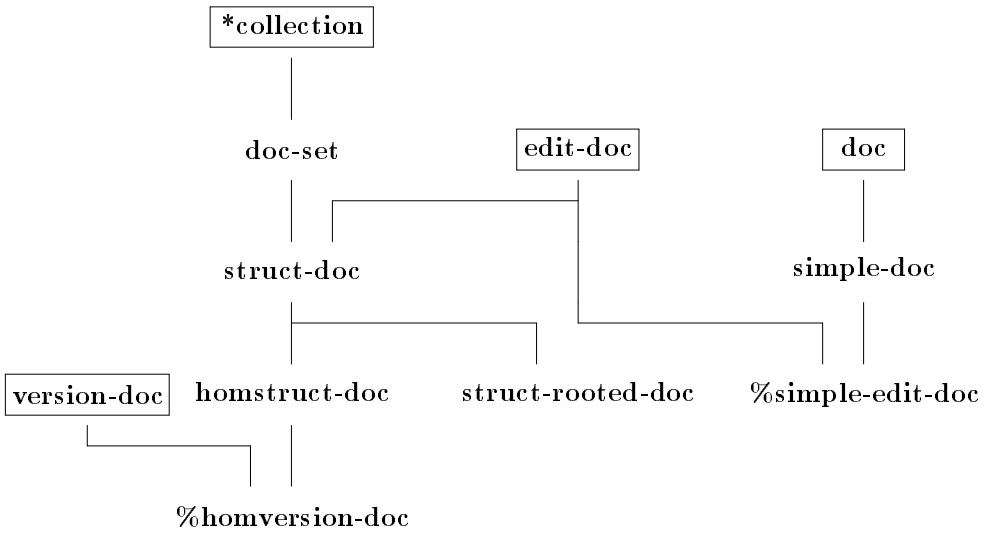

Figure 6.3: The Types for Specifying the Structure of Documents

6.4 Document Structure

The current TACTS version defines two kinds of document structures. A *simple* document has as content one unstructured contiguous part. For a *structured* document the content consists of several subdocuments which are themselves represented as structured or simple TACTS documents. Other document structures could be implemented e.g., to support multimedia documents or documents with a structure not represented by a set of TACTS documents. Figure 6.3 depicts the types defined in this section.

6.4.1 Simple Documents

A simple document is characterized by the fact that the whole content can be loaded in a single emacs buffer. This implies that it has no additional structure which is known by the TACTS system. However, specializations can be defined which interpret the content in particular ways.

Simple documents are defined by the type **simple-doc**. It is a specialization of type **doc** and defines the two additional methods **load-doc** which loads the content into an emacs buffer, and **store-doc** which replaces the content by the content of an emacs buffer. Both methods take a buffer name as argument. The method bodies are not implemented in type **simple-doc** because their implementation depend on the actual storage mechanism used for the document content. However, the methods are used as a common interface to implement all other methods that use the content in some way. These are in particular all operations on abstract documents. The methods for document editing are implemented by

the combination supplement type **%simple-edit-doc**.

The methods for the document operations all use a common standard content buffer which is document specific. The type **simple-doc** defines the two methods **get-contentbuffer** which creates the buffer if necessary and loads the current content in it, and **save-contentbuffer** which writes the standard buffer's content back into the document.

The method **start-edit-doc** gets the standard content buffer and offers it to the user for editing. The method **stop-edit-doc** saves the standard content buffer. The method **erase-doc** erases the standard content buffer and saves it. The methods **init-doc** and **fix-doc** are unchanged with respect to type **doc**.

The method **export-doc** is implemented as follows. First it gets the standard content buffer. The further processing depends on the export system used. One additional export system "pipe" is defined. For the "pipe" system the address must be an (operating system) command. The command is executed and the buffer content is piped into it.

The method **import-doc** is implemented by first filling the standard content buffer in a system dependent way and then saving it. For the predefined import system "export", the content is simply copied from the standard content buffer of the source document, if that was a **simple-doc**. Two additional import systems are defined: "emacs" and "pipe". For the "emacs" import system the address directly is the name of an emacs buffer. This system allows to import the content of an arbitrary buffer manually. For the system "pipe" the address must be an (operating system) command. The command is executed with the standard content buffer as output medium. This system allows to import the output of any command as document content.

6.4.2 Structured Documents

The content of a structured document is a set of documents. Structured documents are implemented using collections (see Section 3.2). The type **doc-set** is a specialization of type ***collection**. It introduces the set named "subdocs" which may be used to hold subordinate document objects. The type **struct-doc** is a specialization of **edit-doc** and **doc-set**. It uses the set "subdocs" to store its document parts. the set "subdocs" which contains all direct document part objects. The new method **show-doc-parts** displays a buffer with a list of all document parts. This list can be used in the same way as the list of subactivities of a **activity-set** as an interactive interface to execute operations on the parts.

For instances of type **struct-doc** the TACTS operations can be used for creating and deleting parts. Semantically, these operations change the content of the structured document. This is the reason for the type **struct-doc** being a specialization of type **edit-doc**. Even if no part itself is of type **edit-doc**, the document as a whole may be edited by creating and deleting parts. Accordingly,

the method **start-edit-doc** is implemented by invoking the method **show-doc-parts**.

The second abstract document operation fully implemented by **struct-doc** is the method **erase-doc**. It works by simply deleting (!) all document parts. This behavior corresponds to the fact that the structure is part of the document content and **erase-doc** does eliminate the current content. If method **erase-doc** would only invoke method **erase-doc** for the parts, it would only erase the "real" content but not the structure. The user can do this by explicitly calling the method **erase-doc** for single parts.

Exporting the document to system "document" is not implemented in the current TACTS version. To do this correctly, the parts must be copied and their attribute values must be copied in a sensible way as well. However, the current version of ELIOOS does not support copying of objects. It is planned to add this feature to ELIOOS. When this is done, copying of structured documents by exporting them to "document" will be implemented as well.

For documents which have a fixed structure, the method **init-doc** can be used to create the necessary document part objects. However, in the general case no concrete document structure is known. Thus the type **struct-doc** inherits this method from type **doc** without changing it.

The type **struct-doc** is incompatible with type **simple-doc**, however it may be combined with all other document features like history or versioning.

The type **struct-rooted-doc** is a specialization of **struct-doc** that defines an additional attribute **Documentroot**. It stores the object number of a document part. This part is interpreted as "document root". The type only defines the method **set-documentroot** to set the attribute. The usage of the value must be defined by specializations of **struct-rooted-doc**. For an example refer to Section 6.6.3.

6.4.3 Homogeneously Structured Documents

For structured documents it is often the case that an operation on the whole document can be implemented by applying the same or a similar operation to all document parts. Structured documents of this kind are called *homogeneously structured*. A typical example is a book which has chapters as parts. An example for a non-homogeneously structured document will be considered in Section 6.6.

Homogeneously structured documents are defined by the type **homstruct-doc** which is a specialization of **struct-doc**. All parts of a **homstruct-doc** are automatically specialized to the type ***homstruct-part**. Together, both types implement the coupling between the whole document and its parts. It is based on the principle that the content of the document consists of all part objects as well as of their contents.

The following consistency conditions for the state of the document are supported:

1. the document is **undefined** iff no part exists,

2. if the document is **fixed**, all parts are **fixed**,

3. if a part is **in-edit**, the document is **in-edit**.

Partly, the first condition is already guaranteed by the type **struct-doc** which deletes all parts whenever the state of the document becomes **undefined**. Additionally, the methods **create-sub** and **delete-sub** defined by type ***collection** are enhanced by performing the corresponding state transitions for the document, if necessary. Specializing a part does a **start-edit-doc** transition. Creating or putting a part does the same, if the state was **existent**. Otherwise the state must be **undefined** and the transition **init-doc** is performed. Deleting performs the transition **erase-doc**, if it was the last existing part. When a document part is moved, this is handled like deleting the part in the source document. This is implemented by enhancing the method **moved** of type ***homstruct-part**.

Note that upward propagation of transitions must be done in a special way. Since it is done in reaction to an action in the document, the operation associated to the transition is already executed and must not be performed by the transition. Thus, upward propagation is done by calling the methods **begin-transit** and **complete-transit** directly. The method **transit-step** is not used and that is why the method **action-transit** is not called (see Section 3.4.2).

The second condition is guaranteed by simply propagating the transition **fix-doc** from the document to all parts. The third condition is guaranteed by propagating the transition **start-edit-doc** from a part up to the document (if necessary) and by propagating the transition **stop-edit-doc** from the document to all parts.

Finally, all state changes of a part which reflect a content change for that part invoke the transition **start-edit-doc** at the document, if necessary. This is the case for the transitions **init-doc**, **import-doc**, and **erase-doc** at parts.

Exporting of homogeneous documents to systems other than "document" is also handled in a special way. In general, exporting is done by exporting all parts in some sequential order to the same system and the same address. The **Changes-system** and **Changes-address** is propagated as well. If all parts are simple documents, this results in exporting exactly those parts which have changed.

6.4.4 Combination with Version Management

The types **struct-doc** and **homstruct-doc** may be combined with **version-doc**. If the parts are of type **version-doc** themselves, a certain coupling between document versions and versions of the parts emerges. For a **struct-doc** this is a very loose coupling, for a **homstruct-doc**, however, the state consistency conditions imply a tighter coupling: every version interval of the document is fully

contained in a version interval of every part with an overlapping version interval. This is guaranteed because every state change that starts or ends a version of a part, either implies a similar state change of the document, or requires that the document is in a state where no version exists.

However, to do a correct version management for type **homstruct-doc**, parts may no longer be deleted, since they belong to the document content as well, and thus information about an older content version would be lost. Correct version management for this kind of documents is implemented by the combination supplement type **%homversion-doc**. It specializes all document parts by adding the type ***homversion-part**. This is a specialization of ***homstruct-part** which adds a new state **old** to the automaton of **doc** together with transitions **setold** from any of the other three states into the **old** state and the corresponding method **setold** which is executed during these transitions.

Whenever the part of a document of type **%homversion-doc** should be deleted, its state is set to **old** instead. This allows it to be still accessed when an old version is required. This behavior is implemented by changing the method **delete-sub** of type **%homversion-doc**.

In the case that the part is itself a **version-doc**, the current version must be ended upon a transition **setold**, because the part can not have a defined version after being deleted. This is done by enhancing the method **setold** of type ***homversion-part** accordingly.

Finally, the exporting of old versions of the document to systems other than "document" is partly implemented by type **%homversion-doc**. If a document version is specified for the method **export**, first all parts are checked if they have a defined version overlapping with the document version. As mentioned previously, in this case there must be a single version interval of the part which contains the whole document version interval. Now, for every part which has a corresponding defined version, this version is exported to the same system and address.

6.5 Document Content

All features for documents described in the previous sections can only be used for structuring and protocoling. To offer real support, the document content itself must be stored by the system. The current version of TACTS supports two variants of content storing for documents: inside the TACTS system itself or in the file system.

Additionally, there are two types describing the *kind* of the content. The two kinds are text and binary content. The kind determines what may be done with the content. Figure 6.3 depicts the types defined in this section.

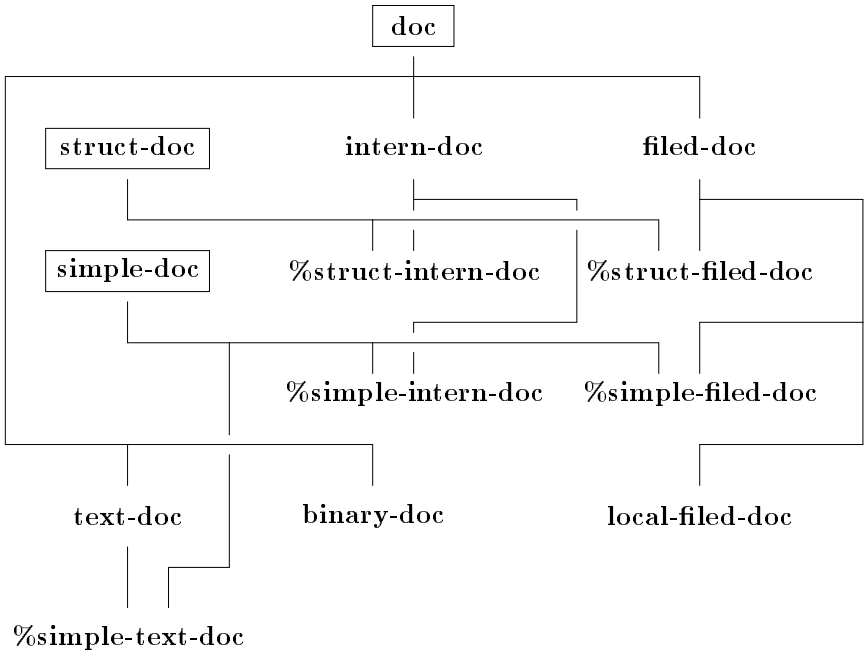

Figure 6.4: The Types for the Content of Documents

6.5.1 Internal Documents

An *internal* document is a document for which TACTS itself stores the content. Internal documents are defined by the type **intern-doc** which is a specialization of **doc**. Depending on the structure of the document, there are two variants of internally stored documents. For simple documents, the combination supplement type **%simple-intern-doc** implements the methods **load-doc** and **store-doc**, thus all operations defined for abstract documents are fully implemented with respect to the content. For structured documents, the combination supplement type **%struct-intern-doc** simply passes the property of being internally stored to the parts by specializing them to type **intern-doc**.

The content storing for simple internal documents is implemented in a way similar to the storing of electronic mails by emacs. A single file is used to store all texts and the texts are separated by special headers. TACTS uses one such file for every object base. Whenever an object of type **%simple-intern-doc** is moved from one base into another, the corresponding text is moved as well, its existence provided.

In the current version of TACTS internal documents do not support version management. The type **intern-doc** may be combined with **version-doc**, however, the export of older versions will not work, because no content is stored for older versions. This will be improved by defining a corresponding combination supplement type.

6.5.2 Filed Documents

An alternative to internal documents are *filed* documents whose content is stored in the file system. Filed documents are defined by the type **filed-doc** which is a specialization of **doc**. It defines the additional attribute **Contentsfile** which stores a file path name.

The file name is synchronized with the document state in the following way: the document is in state **undefined** iff no file name is specified. The file name can only be set or changed during a transition **init-doc** or **import-doc**. Erasing the document does not erase the file name, thus the same name may be used upon the next initialization or import. The additional method **get-contentsfile** simply returns the file name.

The type defines an additional export system "command". For this system the address must be an operating system command containing one or more stars (*). All stars are replaced by the content file name and the command is executed.

The type defines an additional import system "file". For this system the address must be a file name. The name is associated as the new content file name with the document.

The type **filed-doc** is incompatible to the type **intern-doc**. In the current version of TACTS, the type **filed-doc**, like **intern-doc**, does not support versions. Filed documents are useful, if they will be processed by other tools which require the content to be in a file like, e.g., a compiler. However, filed documents have the disadvantage that they may be changed or erased outside of TACTS. In this case TACTS has no information about the change and thus it can not be guaranteed that the state maintained by TACTS actually represents the current state of the document contents.

6.5.3 Filed Documents and Structure

A filed document may be a structured document or a simple document. The first case is defined by the type **%struct-filed-doc** which is a combination supplement type used for the combination of the types **filed-doc** and **struct-doc**. The second case is defined by the type **%simple-filed-doc** which is a combination supplement used for the types **filed-doc** and **simple-doc**.

The type **%simple-filed-doc** interprets the file name as the name of a plain file which stores the content of the document. Like for **%simple-intern-doc**, the methods **store-doc** and **load-doc** are implemented, thus offering the full content manipulation features defined for abstract documents. The methods are implemented by directly using the emacs operations for loading and storing a file.

The type **%struct-filed-doc**, instead, interprets the file name as the name of a directory. This directory is used in a number of ways as default place to work with the document. Additionally, the type **filed-doc** is added to all parts.

The specialization **local-filed-doc** of **filed-doc** can be used for parts of a

%struct-filed-doc for which the file resides in the directory of the structured document. The type **local-filed-doc** only stores the path name relative to the directory of the containing document. It defines the additional parameter **Absolute** for the method **get-contentsfile**. If this is set to TRUE, the absolute pathname of the file is returned, otherwise **get-contentsfile** returns the relative pathname.

Additionally, **local-filed-doc** treats the address of an import from system "file" to be relative to the document's directory, if it is no absolute path name. When a **local-filed-doc** is exported to system "command", the command is executed in the directory of the document, and the relative pathname is inserted.

6.5.4 Document Content Kinds

The type **text-doc** is used to specify a document for which the content is in plain text format, i.e., coded as a sequence of characters in the host computer's code used for characters, such as ASCII or EBCDIC. The type is a specialization of type **doc**.

Two additional export systems are defined by type **text-doc**: "screen" and "printer". The combination supplement type **%simple-text-doc** implements these systems for combinations of the types **simple-doc** and **text-doc**. For the "screen" system the address argument is ignored and the content buffer is simply displayed in read only mode. For the "printer" system the address must be the name of a printer. The buffer content is printed as plain text with the help of the emacs line printer interface.

The type **binary-doc** is used to specify a document for which the content is in some binary format. This may be a text as well, however it can not be read or printed directly. It must be processed in some way to use it. See Section 6.6.3 for an example.

6.6 Processable Documents

Often, electronic documents stored in the computer are also processed in some way by the computer. Processing a document produces another form of the content. Usually, the processed form differs from the source in the operations which can be applied to it. An example for document processing is the processing by text processing systems like TEX or troff. Another example is the processing of program sources by a compiler.

Figure 6.5 depicts the types defined in this section.

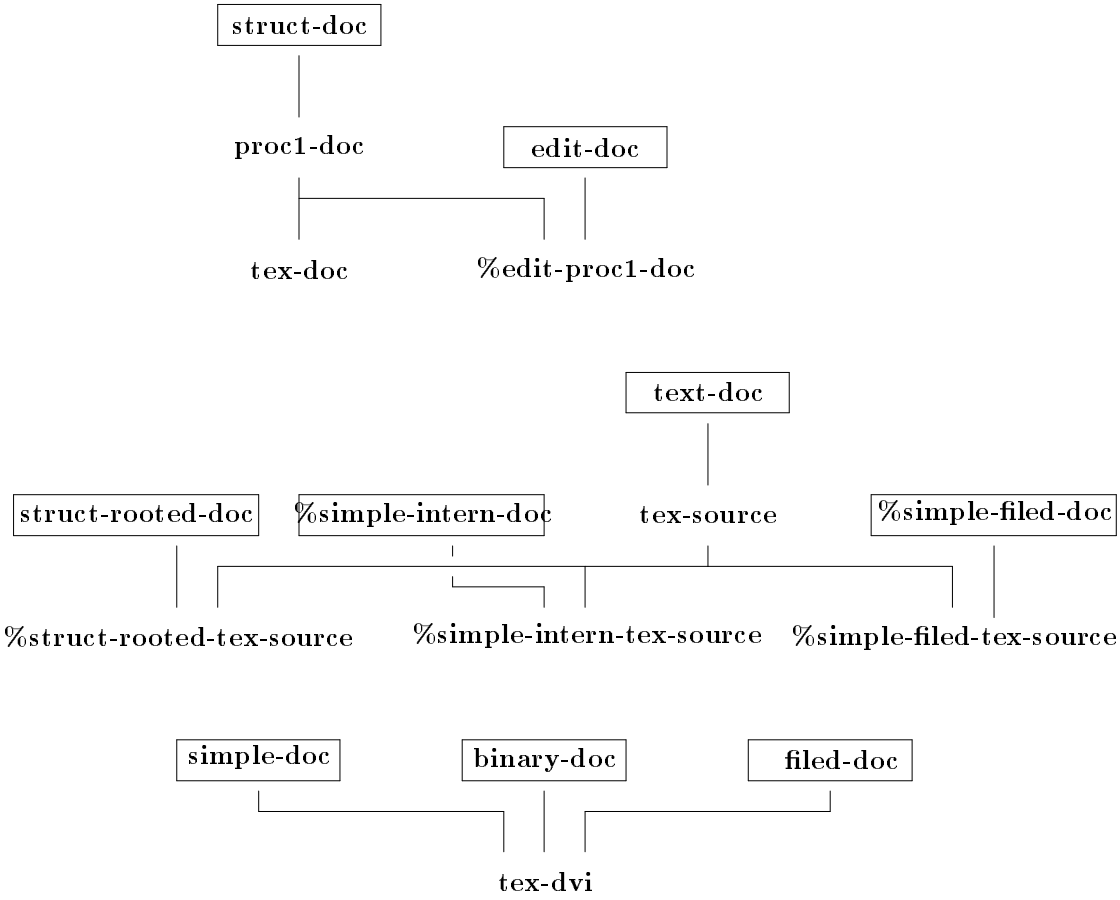

Figure 6.5: The Types for Processable Documents

6.6.1 One-Step Processable Documents

In the current version, TACTS supports processing of documents in a single step. This is defined by the type **proc1-doc**. The combination supplement type **%edit-proc1-doc** is used for the combination of type **proc1-doc** with type **edit-doc**.

Note that document processing can also be achieved by exporting it, e.g., by exporting a simple document to the system "pipe", or by exporting a filed document to the system "command". However, in this case the result of the processing is not automatically accessible to TACTS and can not directly be processed further using TACTS.

The type **proc1-doc**, instead, represents the result of the processing step by a TACTS document as well. In general, a **proc1-doc** has exactly two parts: the source which is a document, and the result of the processing step which is a document as well, but may have types different from those of the source. This implies that a **proc1-doc** is a non-homogeneously structured document and the type **proc1-doc** is defined as a specialization of type **struct-doc**. In a similar way it would be possible to support documents which are processed in several different steps. The one-step case was chosen for simplicity and multi-step extensions will be provided in future TACTS versions.

Upon initialization of a **proc1-doc** the number of parts is already known and the parts are immediately created. For both parts the type **doc** is used. Specializations of type **proc1-doc**, which know more about the kind of the source or the processed part, can add any types which are required in their case.

Remember that erasing a **struct-doc** deletes all its parts. This behavior is inherited by **proc1-doc** as well.

We define the processing step to be the only way of producing the processed part. Thus the methods **init-doc**, **import-doc**, **start-edit-doc**, and **stop-edit-doc** are implemented for the document by propagating them to the source part.

Additionally, the type **proc1-doc** defines the new method **process-doc** which performs the processing step itself. It is implemented by first exporting the source to some export system and afterwards importing the processed part from some import system. The systems and addresses to be used must be specified by the respective specializations of **proc1-doc**. If the processed part was not in state **undefined** it is erased before importing the process result.

Exporting a **proc1-doc** is implemented as follows. If the last change of the source happened after the last change of the processed part, the method **process-doc** is called. Then the export operation is simply propagated to the processed part.

The user is not restricted to the operations defined in this way. Like for any **struct-doc** he/she can also work directly on the source or the processed part. In particular, it is still possible to export the unprocessed source by directly

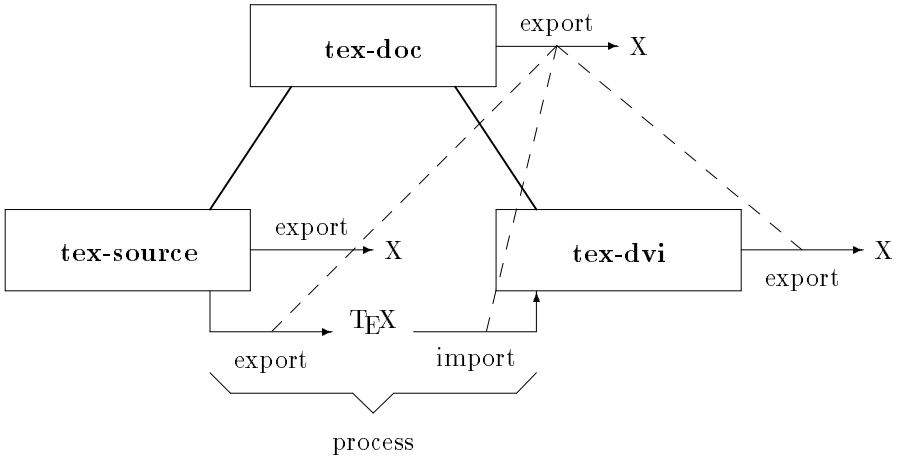

Figure 6.6: Instances Comprising a TEX Document

exporting the source part of the document.

6.6.2 History and Versions for Processable Documents

When a **proc1-doc** is processed or exported, the processed part usually changes. However, this does not mean a real change of the document and thus no state transitions for the document should be performed, like it is done for a homogeneously structured document when a part changes. On the other hand, changes in the source should be registered as changes of the document as a whole.

To achieve this, the type ***homstruct-part** is automatically added to the source part object. Additionally, the type **proc1-doc** propagates all those events to the source part which are propagated by **homstruct-doc** to all parts. This implies that the history and the versions of the whole document mainly reflect the history and versions of the source. Note that the export of old versions automatically works for the whole document, when it is implemented for the source. In this case the method **process-doc** processes the old version by exporting the corresponding version of the source to the processing system.

6.6.3 TEX Documents

As an example for processable documents, TACTS defines types for working with TEX. The processing of the source text by the TEX program resulting in a file in DVI format is modeled by the specialization **tex-doc** of type **proc1-doc**. The source part always has the type **tex-source**, the processed part has the type **tex-dvi**. Figure 6.6 depicts an instance of type **tex-doc** with the subordinate instances of types **tex-source** and **tex-dvi**.

The type **tex-source** is a specialization of the type **text-doc**. It defines the additional export system "tex". The address is used to specify the TEXvariant to be used, such as LATEX or AMSTeX.

The implementation of the export system "tex" depends on the kind of structure and content storing of the TEX source document. The current TACTS version defines variants for simple documents (internal or filed) and for rooted structured documents by defining corresponding combination supplement types.

For simple internal tex sources, a fixed directory is used to run the TEX program and store the resulting DVI-files. Upon each export to system "tex" a unique file name is created and the document content is written to that file in the directory. Then the TEX program is run on that file. This behavior is defined by the combination supplement type **%simple-intern-tex-source**.

For simple filed tex sources, the TEX program runs directly on the content file in the directory where the content file resides.This behavior is defined by the combination supplement type **%simple-filed-tex-source**.

For rooted structured documents, the TEX program is executed on the "root part" (see type **struct-rooted-doc**, Section 6.4.2) by exporting the root part to "tex". The root part must be a **tex-source**. Depending on its kind, the TEX program either runs in the directory associated with the type **%struct-filed-doc** or somewhere else. This behavior is defined by the combination supplement type **%struct-rooted-tex-source**.

The type **tex-dvi** is a specialization of the types **binary-doc**, **simple-doc**, and **filed-doc**. This reflects the fact that TEX always produces its result in a binary format in one part, stored in a file.

As additional supported import system the type **tex-dvi** defines the system "tex". Importing from system "tex" is implemented by importing the DVI-file produced by the last export of a tex source to system "tex". Accordingly, processing the TEX document as a whole is implemented by exporting the source to system "tex" and afterwards importing the DVI part from system "tex". This is depicted in figure 6.6.

Two additional export systems are defined by type **tex-dvi** (and can thus be used for objects of type **tex-doc**): "screen" and "printer". The "screen" system can be used for previewing, the "printer" system prints the DVI-file on the printer specified as address. Note that both the source and the DVI-file can be printed or viewed by exporting them to "printer" or "screen". Because of their different types, these operations are executed in a totally different way.

As mentioned previously, exporting the whole TEX document is implemented by processing it, if necessary, and then exporting the DVI part (see Section 6.6.1). Figure 6.6 depicts the three possibilities for exporting to an export system X. The implementation of the export of the whole document by operations on the subordinate instances is shown by the dashed lines.

In the current TACTS version, it is assumes that TACTS is running under the X window system and that a postscript printer is available. Exporting a

DVI-file document to "screen" is directly implemented using the xdvi command, exporting to printer is implemented using the dvips driver. In future versions it is planned to be more flexible and define specializations for different ways of exporting DVI-files. In particular, exporting to postscript will be modeled by an additional processing step and the resulting postscript document will become a third part of the **tex-doc** which can itself be previewed, printed, or used in some other way.

Chapter 7

Communication

This chapter describes the support of communication activities. The general principle is a mechanism to transmit TACTS documents over electronic media, such as electronic mail or USENET news.

7.1 Supporting Communication

First we introduce the mechanisms for transmitting TACTS documents over electronic media. This is implemented by types that specialize the types for documents defined in Chapter 6. Figure 7.1 depicts the corresponding part of the TACTS type hierarchy. For simplicity, only the types for sending via electronic mail are included. Analogous types exist for receiving and for USENET news.

7.1.1 Communication Environments and Addressing

The basic concept of supporting communication by supporting the transmission of TACTS documents is a very general approach. In the current version of TACTS specific support for two systems is implemented. These are the electronic mail system and the USENET news system. They were chosen since they are readily available under UNIX and since interfaces for their access exist in GNU emacs.

In both systems additional information is necessary to specify the details of a transmission. For an electronic mail at least the recipient must be given. For a news article the newsgroup to be used must be specified. Both systems use the same mechanism for this kind of information. A set of headers is prepended to the content. Each header starts with a keyword and has a string as value. TACTS supports this kind of information by types for managing a header set.

There is one important difference between the two systems in respect of addressing. Electronic mail typically is transmitted between two agents. Both agents are identified by an email address of the form "user@host". In the news system articles are not transmitted to a single agent. Instead, they are posted in

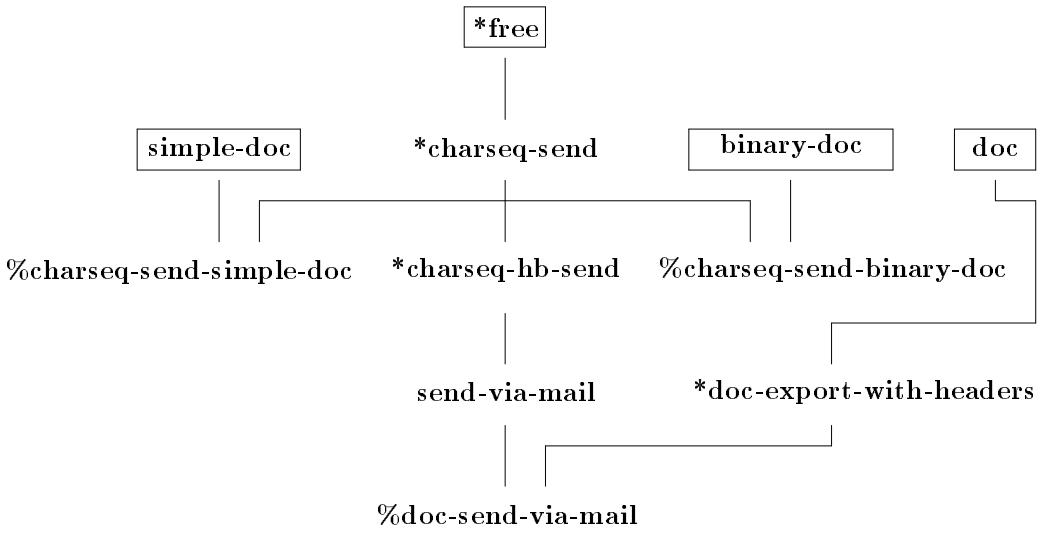

Figure 7.1: The Types for Sending Documents via Electronic Mail

the form of a broadcast with no specific recipient given. Upon receiving a news article, however, the posting agent may be determined by its email address which is included in the article. Thus, receiving electronic mail and news articles, as well as sending electronic mail can be modeled by using the type **two-agent-trans** introduced in Section 4.5.2. Posting a news article only uses the types **send-info** and **by-single**.

TACTS supports electronic mail addressing in a special way. The user does not specify the address directly. Instead, the user specifies an agent (see Section 4.2) and the address is stored at the agent and retrieved by the system. This has two advantages. The addresses are associated with other information about the agent, such as the name, and this information can be retrieved by the system as well. On the other hand, the user need not remember the address of an agent for sending a mail.

The email address for agents is implemented by the specialization **agent-with-email** of type **agent**. It defines the key "emailadr" for the method **agent-info**. When given this key the method returns the email address of the agent.

7.1.2 The Transmission of Text Pieces with Headers

Both in electronic mail and in the news system, the content of the transmitted information must be a contiguous piece of plain text. We use this property as a characteristics for the most general types defined in this section, the designer concept types ***charseq-send** and ***charseq-receive**. These types describe every object that can be converted to or from a contiguous plain text (a "character sequence") which may be transmitted over any medium that supports this kind of

content. Currently, specializations of these types for two kinds of documents and the two communication systems mail and news are implemented. The mechanism can easily be extended by defining additional specializations.

The type ***charseq-send** simply defines the method **charseq-prepare**. It returns a list of emacs buffers. Every buffer contains plain text and will be transmitted in one part. Analogously, the type ***charseq-receive** defines the method **charseq-process**. It takes a list of emacs buffers as argument. Every buffer contains one part which was received.

Both methods have an empty body. They are used as a common interface between specializations for specific transmission media and specializations for specific kinds of transmitted objects. Thus it is possible to combine any type for a specific kind of transmitted object with any type for a transmission medium to form a type which implements the transmission of objects of the specific kind over the specific medium.

The management of a header set is implemented by the types ***charseq-hb-send** and ***charseq-hb-receive**. They are specializations of the types ***charseq-send** and ***charseq-receive**, respectively. They define the additional methods **charseq-hb-prepare** and **charseq-hb-process**, respectively. Thus, the header management is independent from both the transmission system used and the implementation of preparing and processing the content.

The method **charseq-hb-prepare** takes a header set as argument. It invokes the method **charseq-prepare** to prepare the content buffers. Then it prepends the given headers to every buffer and returns the list of buffers modified in this way. The method **charseq-hb-process** takes a buffer list as argument. It parses every buffer and extracts the headers from it, leaving only the body. Then it invokes the method **charseq-process** to process the body parts. It returns the header set as result.

7.1.3 The Interface to the Transmission Medium

A type that implements a specific transmission medium defines methods for sending or receiving, respectively. The method for sending first invokes the method **charseq-prepare** or **charseq-hb-prepare** to prepare the content for sending. Then it sends out the content of the returned buffers in a system specific way. The method for receiving first receives one or more parts in a system specific way and puts them into buffers. Then it invokes the method **charseq-process** or **charseq-hb-process** with the list of these buffers.

Accordingly, the types **send-via-mail**, **receive-via-mail**, **send-via-news**, and **receive-via-news** are defined for sending and receiving electronic mail and news. They define the methods **send-mail**, **receive-mail**, **send-news**, and **receive-news**, respectively, and implement them in the way described above. The methods for sending take as single argument a header list which is passed to method **charseq-hb-prepare**. The methods for receiving take no argument.

They return the header list extracted by method **charseq-hb-process**.

Note that before a mail or news can be received by TACTS, the user has to select it using the corresponding GNU emacs subsystem that provides an interface to the communication system. In this way the user can still decide which mails or articles should be processed by TACTS and which should be processed in other ways. However, more support for automatic reception of mail and news articles will be provided in future TACTS versions.

For documents, sending and receiving the content is integrated in a special way into the existing document operations. Sending is done by exporting the content, receiving is done by importing the content. Thus it is possible to use all other features of these operations, such as exporting changes or exporting older versions, in combination with electronic transmission.

To enable the specification of a header set for export and the retrieving of the header set upon import, the types ***doc-export-with-headers** and ***doc-import-with-headers** are used. The former defines the additional argument **Headers** for the method **export-doc**, the latter allows the returning of the header set by the method **import-doc**.

The combination supplement type **%doc-send-via-mail** is used for the combination of the types **doc** and **send-via-mail**. It is a specialization of type ***doc-export-with-headers** and defines the additional supported export system "email". The destination agent is specified in the parameter **Address** of the method **export-doc**. The method maps the given agent to its email address (see Section 7.1.1), prepares the corresponding header "To" and adds it to the header set. The resulting header set is passed to the method **send-mail** which is called to send out the document content.

Note the difference between headers and body in the case of documents. The headers are not part of the document content. Instead, they are specified as arguments to every export operation to "email". Thus, the same document may be transmitted several times using different header sets.

In a similar way the types **%doc-receive-via-mail**, **%doc-send-via-news**, and **%doc-receive-via-news** define and implement the import system "email" and the export and import systems "news" respectively. However, in both cases of import systems the parameter **Address** of method **import-doc** is ignored and for the export to "news" the parameter **Address** of method **export-doc** is used to specify the newsgroup instead of the recipient.

7.1.4 The Interface to the Transmitted Object

The types described in Section 7.1.2 implement the interface to the transmission medium. Now we will describe the types which implement the interface to the transmitted object, in particular, to the document content. These types are specializations of the types ***charseq-send** and ***charseq-receive**, respectively, which implement the methods **charseq-prepare** and **charseq-process**.

First we define the interface to simple documents. This is easy, since the content of a simple document can be loaded into a single emacs buffer and can be sent in one part. Thus the method **charseq-prepare** simply creates a buffer list containing one buffer and copies the content from the standard content buffer into the single buffer in the list. This is implemented by the combination supplement type **%charseq-send-simple-doc** used for the combination of the types ***charseq-send** and **simple-doc**. Analogously, the type **%charseq-receive-simple-doc** implements the method **charseq-process** by copying the content of the first buffer in the given list into the standard content buffer and saving it.

Next we handle the content kind. Documents of type **text-doc** have as content plain text which can be directly handled by the charseq types. The content of documents of type **binary-doc**, however, must first be converted to text before it may be transmitted using the charseq types. This is done using the UNIX uuencode/uudecode mechanism for coding binary data as text. The combination supplement type **%charseq-send-binary-doc** used for the combination of the types ***charseq-send** and **binary-doc** enhances the method **charseq-prepare** by applying uuencode to every buffer in the list. The combination supplement type **%charseq-receive-binary-doc** used for the combination of the types ***charseq-receive** and **binary-doc** enhances the method **charseq-process** by applying uudecode to the content of every buffer in the given list.

Note, that the user interface for sending and receiving documents is defined by the types **%doc-send-via-mail**, **%doc-receive-via-mail** and the corresponding types for the news system. Thus, it does not depend on the document content form. Text documents and binary documents both are sent by exporting them to system "email" or "news".

It is planned to support transmission of other documents as well. Homogeneously structured documents can easily be supported by either transmitting every component as a separate part, or by generating a structured text containing all parts and transmitting this text in one piece. The latter can be implemented using an archive mechanism like UNIX shar.

7.1.5 Communication and Document History

Using the mechanisms described in the previous sections, a document can be sent or received several times. Since all transmissions are implemented using the document operations export and import, every transmission can be protocoled as a subactivity. This is done using the history mechanism introduced in Section 6.2. During every export or import transition a subactivity of type **basic-activity** is created and completed. Thus it stores the time point of performing the transition.

However, additional history support is added for the special cases of sending and receiving the document content. Using combination supplement types, the combination of **send-via-mail** and **history-doc** adds the types **send-info**, **two-agent-trans**, **with-headers** (see Section 7.2.2) and **by-self** to the transition

activities created for exports to system "email" and sets the attributes **Medium**, **Partner**, and **Headers**.

Analogously, the combination of type **history-doc** with the types **receive-via-mail**, **send-via-news**, and **receive-via-news** is handled by storing all relevant information of the transmission at the transition activity instance.

7.1.6 Communication and TEX Documents

As an example for the transmission of documents consider the TEX documents described in Section 6.6.3. Both the source part and the processed part of a document may be sent by exporting it to system "email" or "news". If the source part is of types **simple-doc** and **tex-source**, it is transmitted in one part as plain text. The processed part is of type **tex-dvi** which contains the types **simple-doc** and **binary-doc**. Thus it is automatically converted to text using uuencode and transmitted in one part.

As mentioned previously, exporting the TEX document as a whole is implemented by first processing it, if necessary, and then exporting the processed part. This also works for sending the document. It is automatically processed by TEX, the resulting DVI-file is converted by uuencode and then sent out to the communication medium. The situation is that of figure 6.6, if the export system X is replaced by the concrete export system "email" or "news", respectively. This is a good example of the advantages of an integrated system like TACTS. If it is carefully designed several single concepts may be combined to yield additional functionality.

7.2 Communication Activities

In Section 7.1 we only handled one aspect of transmitting documents. The document was the main object, the transmission activities played the role of subactivities. This situation is typical for large documents which are changed and re-transmitted several times. If, however, a short information is transmitted, e.g., via electronic mail, the focus of interest typically is the single transmission activity. The document is secondary and should be local to the activity.

In the current section we define types supporting this kind of communication activities. Figure 7.2 depicts the corresponding part of the type hierarchy. For simplicity, only the types for electronic mail are included. Analogous types for the news system exist.

7.2.1 Activities for Document Transmission

The first step is to introduce activities with an associated document. This is done by type **with-doc** which is a specialization of type ***with-ref**. It defines a new

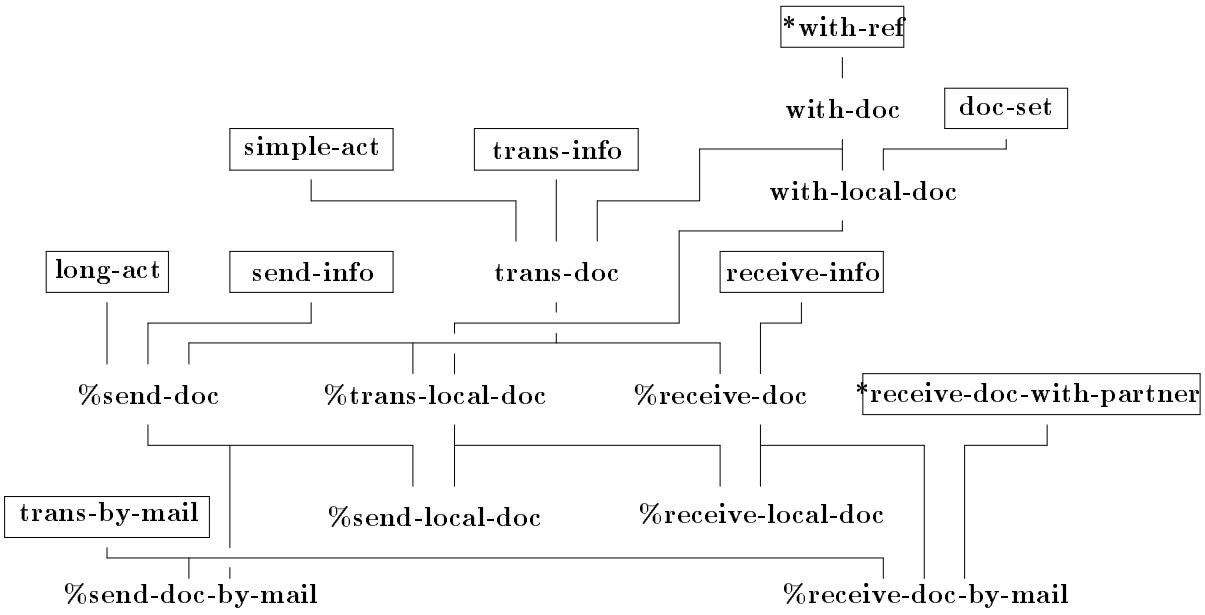

Figure 7.2: The Types for Activities which Transmit a Document

attribute **Document** which is used to store a reference to a document object. This type is further specialized by making the document a subobject of the main activity. The corresponding type **with-local-doc** combines the types **with-doc** and **doc-set** and enhances its method **created** in order to create an object of type **doc** as subobject and set the attribute **Document** to a reference to this document.

Both types can be used for any activity which involves a single subordinate document. The type **trans-doc** is the specialization for activities that transmit the document. It is a specialization of the types **trans-info**, **with-doc**, and **simple-act**. It couples the transition **complete** of the activity with a transition (either import or export) of the referenced document, using the value of the attribute **Medium** as system.

There are combination supplement types for the combination of **trans-doc** with the types **send-info** and **receive-info**. The type **%send-doc** sets the transition used at the document to **export-doc**, the type **%receive-doc** sets it to **import-doc**.

Next, types for the transmission of local documents are introduced. Since the document is local to the activity, it makes sense to tie the transitions of the document's state even more to transitions of the activity's state than it is done by type **trans-doc**. In detail, since the document is only involved in the single activity of sending or receiving it, it should be fixed after import or export, respectively, to the communication system. This is implemented by the combination supplement type **%trans-local-doc** used for the combination of the types **trans-doc** and

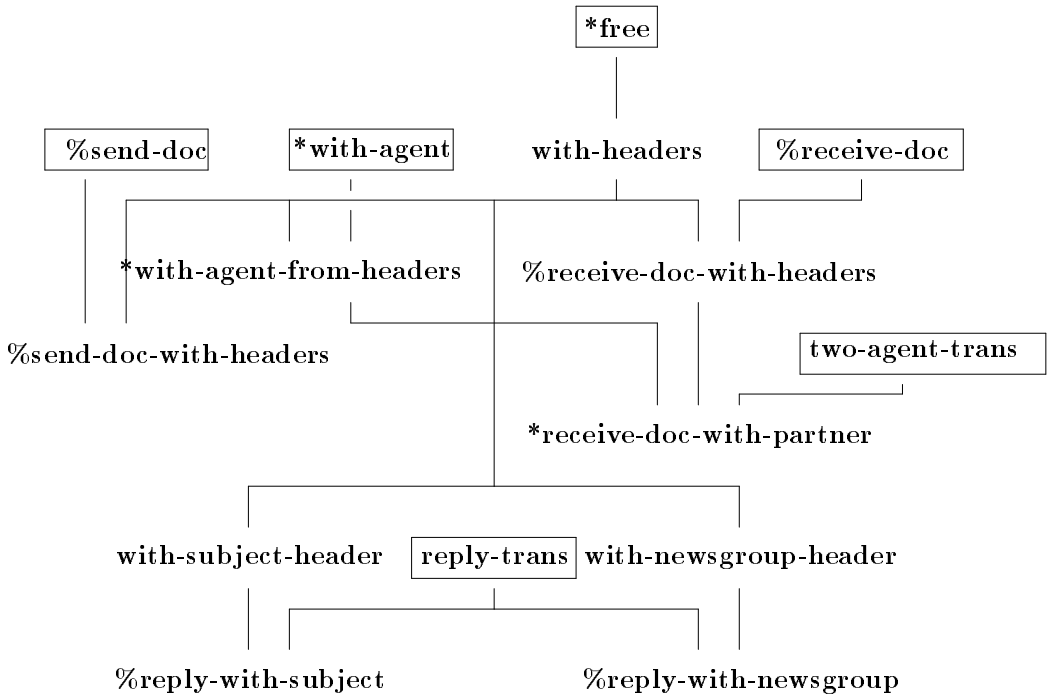

Figure 7.3: The Types for Transmission Activities with Header Management

with-local-doc.

Before a document can be sent out, it must be written (in contrast to documents which are received). If the document is local the writing of the document should be a part of the transmission activity. Thus the combination supplement type **%send-local-doc**, used for the combination of the types **trans-doc**, **send-info**, and **with-local-doc**, uses the additional type **long-act** to introduce the additional state **active** (see Figure 3.5). During the transition **activate** it performs the transition **init-doc** at the document.

Thus instances of type **%send-local-doc** can be used as follows. After their creation they are in state **planned**, the associated document exists and is in state **undefined**. When the user executes the transition **activate**, the document content is initialized. Optionally, the user may now edit the document one or more times. Then the user executes the transition **complete**, and the document content is fixed and sent out.

7.2.2 Header Management for Transmission Activities

Next, types for header management as part of transmission activities are introduced. They are depicted in Figure 7.3. The type **with-headers** defines the attribute **Headers** which stores a header set. Additionally, it defines the method **set-header** for interactive entering or changing a header.

There are two combination supplement types for the combination of **with-headers** with the types **%send-doc** and **%receive-doc**, respectively. The type **%send-doc-with-headers** passes the headers to the method **export-doc** at the document during the transition **complete**. The type **%receive-doc-with-headers** sets the headers to the header set returned by the method **import-doc** during transition **complete**.

The type **with-headers** can be used as basis for several kinds of automatic header manipulation. To demonstrate this, the current version of TACTS defines three such kinds: managing a subject header, managing a newsgroup header, and determining the partner agent from the headers.

The type **with-subject-header** automatically creates the header "Subject" which is commonly used for electronic mail and news articles. When this type is combined with type **reply-trans**, the combination supplement type **%reply-with-subject** enhances the method **set-ref** in the following way. Whenever the reference **Reply-of** is set, the subject of the referenced activity is retrieved and the subject of the reply activity is set to the concatenation of "Re:" and this subject. This is the standard way of setting the subject of a reply to electronic mail and news articles.

The type **with-newsgroup-header** analogously creates the header "Newsgroup" which is used for news articles. For the combination of type **with-newsgroup-header** with type **reply-trans** the combination supplement type **%reply-with-newsgroup** copies the newsgroup header whenever the reference **Reply-of** is set.

The type ***with-agent-from-headers** is a designer concept type that introduces the method **set-agent-from-headers**. It takes as arguments the agent attribute to be set and a parsing function. The parsing function has to look at the headers and return a predicate for searching and information for creating an agent. The method **set-agent-from-headers** first calls the parsing function. Then it invokes the method **search-member** (see Section 3.2.3) to search all agent sets on the path to the base's root using the predicate returned by the parsing function. If an agent is found it is used to set the agent attribute. Otherwise, a new agent is created in the nearest agent set, using the information returned by the parsing function, and the agent attribute is set to a reference to the new agent.

As a specialization of the types ***with-agent-from-headers**, **two-agent-trans**, and **%receive-doc-with-headers** the type ***receive-doc-with-partner** is defined. Upon transition **complete** it sets the partner agent from the headers after receiving the document and setting the headers.

7.2.3 Sending and Receiving Mail and News

Activities of information transmission by electronic mail are described by the type **trans-by-mail**. It is a specialization of the types **two-agent-trans** and

with-headers. Upon creation of an instance it sets the attribute **Medium** to "email".

There are the two combination supplement types **%send-doc-by-mail** and **%receive-doc-by-mail** for combining the type **trans-by-mail** with the types **%send-doc** and **%receive-doc**, respectively. If necessary, both types automatically add the type **send-via-mail** or **receive-via-mail** to the referenced document during the transition **complete** to make the mail transmission mechanism available for the document.

Additionally, the type **%send-doc-by-mail** passes the partner agent in the parameter **Address** to the method **export-doc** at the document. The type **%receive-doc-by-mail** is a specialization of type ***receive-doc-with-partner**. It uses a special header parsing function that tries to extract the email address and the name from the header "From". Thus, every instance of the type automatically searches or creates the partner agent after receiving a mail.

Analogously, activities of information transmission using the news system are described by the type **trans-by-news**. It is a specialization of the types **with-newsgroup-header**, **trans-info**, and **by-single**. The combination supplement types **%send-doc-by-news** and **%receive-doc-by-news** automatically add the type **send-via-news** or **receive-via-news** to the referenced document during the transition **complete** to make the news transmission mechanism available for the document.

Additionally, the type **%send-doc-by-news** passes the value of the header "Newsgroup" in the parameter **Address** to the method **export-doc** at the document. The type **%receive-doc-by-news** is a specialization of type **two-agent-trans** and ***receive-doc-with-partner**. It uses the same parsing function as type **%receive-doc-by-mail**, since news articles contain a header "From" with the same syntax as for mails.

Note the difference between mail and news. For the news system, only the activity of receiving is of type **two-agent-trans**. Instead of the partner agent, the activity of sending uses the "Newsgroup" header to specify the address for export.

Finally, some user packet types are introduced, three for each transmission system. They are listed in the Table 7.1. The first line gives the user packet types. Under every packet type the user concept types are listed which are combined to form the packet type.

The packets form an easy-to-use mechanism for handling single mails and single news articles in a way similar to typical user interfaces to both systems. As an example, an instance of type **send-mail** behaves as described for type **%send-local-doc** in Section 7.2.1, it uses the agent mechanism to specify the communication partner, it records the time of activation and that of sending, it stores the content of the mail or news article internally in TACTS, and it supports printing, viewing, and other kinds of export for the content at any time.

As a more complex application, these types are used in requests (see Chap-

send-mail	reply-mail	receive-mail
trans-by-mail	trans-by-mail	trans-by-mail
trans-doc	trans-doc	trans-doc
with-own-doc	with-own-doc	with-own-doc
timed-state	timed-state	timed-state
by-self	by-self	by-self
send-info	send-info	receive-info
with-subject-header	with-subject-header	
init-partner	reply-trans	

send-news	reply-news	receive-news
trans-by-news	trans-by-news	trans-by-news
trans-doc	trans-doc	trans-doc
with-own-doc	with-own-doc	with-own-doc
timed-state	timed-state	timed-state
by-self	by-self	by-self
send-info	send-info	receive-info
with-newsgroup-header	with-newsgroup-header	with-newsgroup-header
with-subject-header	with-subject-header	
	reply-trans	two-agent-trans

Table 7.1: User Packet Types for Electronic Mail and USENET News

ter 5). The type **mail-request** is a specialization of the type **request**. It adds the types common to **send-mail** and **receive-mail** to the transition activities that model the transmissions of the question and the answers.

Chapter 8

Conclusion

The report presented the TACTS activity support system in form of a detailed description of it type hierarchy. The Chapters 3 – 5 introduced types for the support of arbitrary activities by structuring and protocoling. Chapter 6 introduced types for activities involving documents, additionally supporting the execution of certain subactivities, such as printing. Chapter 7 introduced types for activities involving communication. They add support of execution for communication via electronic media such as electronic mail.

The object-oriented mechanism used as a basis has proven very useful in the design and implementation of this flexible and easily extendible type hierarchy. The fine-grained multiple inheritance mechanism facilitated the definition of every aspect in a single type. The type combination at runtime offers the necessary flexibility to the user.

In its current state, TACTS is only the starting point for a real application. However, it can already be used successfully to support some specific kinds of work. The further evolution of the system should be driven by two directions: adding more basic features, such as better support for time planning, and adding support for specific domains, such as collaborative editing of documents.

The version of TACTS described in this report is fully implemented and is freely available. To obtain the system, contact the author.

Bibliography

[1] Fish, R.S., Kraut, R.E., Leland, M.D.P.: Quilt: A Collaborative Tool for Cooperative Writing. In: Allen, R.B. (ed.): Proc. ACM SIGOIS/IEEE TC-OA Conf. on Office Information Systems, Palo Alto, CA, Mar. 1988. ACM SIGOIS Bulletin, **9**:2&3, pp. 30–37

[2] Hennessy, P., Kreifelts, T., Ehrlich, U.: Distributed Work Management: Activity Coordination within the EuroCoOp Project. Computer Communications, Butterworth-Heinemann, Ltd. **15**:8, 477–488 (Oct. 1992)

[3] Johnson, Paul, Rees, Ceri: Reusability Through Fine-grain Inheritance. Software – Practice and Experience **22**:12, 1049–1068 (Dec. 1992)

[4] Karbe, Bernhard: Wirklichkeitsgerechte Koordinierung kooperativer Bürovorgänge. In: Friedrich, J., Rödiger, K.-H. (eds.): Computergestützte Gruppenarbeit (CSCW). 1. Fachtagung 1991, Bremen, Stuttgart, 1991. Teubner, pp. 207–219

[5] Kreifelts, T., Hinrichs, E., Klein, K.H., Seuffert, P., Woetzel, G.: Experiences with the DOMINO office procedure system. In: Bannon, L., Robinson, M., Schmidt, K. (eds.): Proc. 2nd Europ. Conf. on Computer-Supported Cooperative Work, Amsterdam, The Netherlands, Sep. 1991. Dordrecht, Boston, London: Kluwer Academic Publ., pp. 117–130

[6] Kuutti, Kari: The concept of activity as a basic unit of analysis for CSCW research. ECSCW'91 Proc. of the 2nd European Conf. on Computer Supported Cooperative Work, Sep 25-27 1991

[7] Medina-Mora, Raul, et al.: The action workflow approach to workflow management technology. CSCW'92: Sharing Perspectives, Nov 1992

[8] Neuwirth, C.M., Kaufer, D.S., Chandhok, R., Morris, J.: Issues in the design of computer support for co-authoring and commenting. Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '90), Los Angeles, California, 1990. ACM Press

[9] Schmid, H.: Mechanismen zur Verwaltung typischer Arbeitsabläufe am Institut für Informatik. Inst. für Informatik, Techn. Univ. München, Munich, Germany, Master's thesis, Nov. 1992

[10] Shepherd, A., Mayer, N., Kuchinsky, A.: Strudel: An Extensible Electronic Conversation Toolkit. Proc. 3rd Int. Conf. on Computer-Supported Cooperative Work, Los Angeles, CA, Oct. 1990. New York: SIGCHI/SIGOIS ACM, pp. 93–104

[11] Winograd, T.: A Language/Action Perspective on the Design of Cooperative Work. Human Computer Interaction **3**:1, 3–30 (1988)

Index

(The letters in brackets denote the identifier sort:
A = attribute, M = method, R = transition, S = state, T = type)

*active-transition (T), 26
*active-transitions (T), 19, 25, 26, 36, 41, 43
*charseq-hb-receive (T), 61
*charseq-hb-send (T), 60, 61
*charseq-receive (T), 60–63
*charseq-send (T), 60–63
*collection (T), 12, 13, 15, 16, 18, 19, 22, 46, 47, 49
*collection-member (T), 13, 15, 16, 19, 23
*doc-export-with-headers (T), 60, 62
*doc-import-with-headers (T), 62
*free (T), 11–13, 16, 19, 33, 60, 66
*homstruct-part (T), 48–50, 56
*homversion-part (T), 50
*receive-doc-with-partner (T), 65–68
*state-member (T), 19, 22, 23
*timed-member (T), 16, 18, 19, 22
*transition-activity (T), 19, 24
*with-agent (T), 30, 66
*with-agent-from-headers (T), 66, 67
*with-ref (T), 12, 30, 33, 64, 65
%charseq-receive-binary-doc (T), 63
%charseq-receive-simple-doc (T), 63
%charseq-send-binary-doc (T), 60, 63
%charseq-send-simple-doc (T), 60, 63
%collection-and-member (T), 13, 15
%doc-receive-via-mail (T), 62, 63
%doc-receive-via-news (T), 62
%doc-send-via-mail (T), 60, 62, 63
%doc-send-via-news (T), 62
%edit-proc1-doc (T), 54, 55
%history-edit-doc (T), 41, 43, 44
%homversion-doc (T), 46, 50
%receive-doc (T), 65–68
%receive-doc-by-mail (T), 65, 68
%receive-doc-by-news (T), 68
%receive-doc-with-headers (T), 66, 67
%receive-local-doc (T), 65
%reply-with-newsgroup (T), 66, 67
%reply-with-subject (T), 66, 67
%send-doc (T), 65–68
%send-doc-by-mail (T), 65, 68
%send-doc-by-news (T), 68
%send-doc-with-headers (T), 66, 67
%send-local-doc (T), 65, 66, 68
%simple-edit-doc (T), 46, 47
%simple-filed-doc (T), 51, 52, 54
%simple-filed-tex-source (T), 54, 57
%simple-intern-doc (T), 51, 52, 54
%simple-intern-tex-source (T), 54, 57
%simple-text-doc (T), 51, 53
%struct-filed-doc (T), 51–53, 57
%struct-intern-doc (T), 51
%struct-rooted-tex-source (T), 54, 57
%trans-local-doc (T), 65
%two-agent-reply (T), 33, 34
%version-edit-doc (T), 41, 45

abort (R), 28, 29
abort-act (T), 28
aborted (S), 28, 29
action-transit (M), 21, 24, 42, 44, 49

activate (R), 20, 22, 28, 29, 37, 38, 66
active (S), 20, 21, 28, 29, 37, 38, 66
activity-set (T), 13, 15, 19, 22, 24, 28, 29, 47
add-trans-act (M), 25
addtime (M), 18, 19
agent (T), 29, 30, 60
agent-info (M), 29, 60
agent-set (T), 30
agent-with-email (T), 60
attach (M), 9

basic-activity (T), 28, 29, 31, 37, 44, 63
begin-transit (M), 21, 22, 49
binary-doc (T), 51, 53, 54, 57, 60, 63, 64
By (A), 31
by-partner (T), 5, 6, 30, 32, 34
by-self (T), 5, 30, 31, 33, 63, 69
by-single (T), 5, 30, 31, 33, 36, 60, 68

cancel (R), 28, 29
cancel-act (T), 28
canceled (S), 28, 29
charseq-hb-prepare (M), 61
charseq-hb-process (M), 61, 62
charseq-prepare (M), 61–63
charseq-process (M), 61–63
complete (R), 20, 22, 27–29, 37, 38, 44, 65–68
complete-transit (M), 21, 22, 49
completed (S), 20, 22, 27–29, 37
Contentsfile (A), 52
create-sub (M), 49
created (M), 7, 8, 24, 25, 65
cuttime (M), 18, 19

define-version-name (M), 45
delete (M), 7
delete-sub (M), 13, 49, 50
delete-version-name (M), 45
deref (M), 12
Descr (A), 12, 33
display (M), 11, 13
display-subs (M), 13
display-versions (M), 45
doc (T), 40–44, 46–48, 50–53, 55, 60, 62, 65
doc-set (T), 46, 47, 65
Document (A), 65
Documentroot (A), 48

edit-descr (M), 12
edit-doc (T), 40–47, 54, 55
erase-doc (M), 41, 42, 44, 47, 48
erase-doc (R), 43, 49
existent (S), 42–44, 49
export (M), 50
export-doc (M), 41, 42, 44, 45, 47, 62, 67, 68
export-doc (R), 43, 65
Export-times (A), 43

filed-doc (T), 51, 52, 54, 57
finish (M), 22, 23
fix-doc (M), 41, 47
fix-doc (R), 43, 49
fixed (S), 42–44, 49
follow-ref (M), 12

get-contentbuffer (M), 47
get-contentsfile (M), 52, 53
go-down (M), 15
go-up (M), 15

Headers (A), 64, 66
history-doc (T), 41, 43, 63, 64
homstruct-doc (T), 46, 48–50, 56

immediate (T), 24, 38
import-doc (M), 42, 44, 47, 55, 62, 67
import-doc (R), 43, 49, 52, 65
in-edit (S), 42–44, 49
inherit-by (T), 5, 30–32, 34, 37
inherit-partner (T), 30, 32, 34, 37
init-by (T), 5, 30–32

init-doc (M), 41, 42, 44, 47, 48, 55
init-doc (R), 43, 49, 52, 66
init-partner (T), 30, 32, 69
intern-doc (T), 51, 52

Last-change (A), 42
load-doc (M), 46, 51, 52
local-filed-doc (T), 51–53
long-act (T), 27–29, 36, 37, 44, 65, 66

mail-request (T), 69
Medium (A), 33, 64, 65, 68
moved (M), 7, 49

named-version-doc (T), 41, 45

old (S), 50

Partner (A), 32, 33, 64
planned (S), 20, 22, 24, 27–29, 37, 38, 66
proc1-doc (T), 54–56
process-doc (M), 55, 56

receive-info (T), 33, 34, 38, 65, 69
receive-mail (M), 61
receive-mail (T), 69
receive-news (M), 61
receive-news (T), 69
receive-via-mail (T), 61, 64, 68
receive-via-news (T), 61, 64, 68
refined-transitions (T), 19, 24, 25
reply-mail (T), 69
reply-news (T), 69
Reply-of (A), 33, 34, 67
reply-trans (T), 33, 34, 37, 66, 67, 69
request (T), 35–38, 69
request/addressee (T), 36, 38
request/initiator (T), 36, 38
reset (R), 28, 29
reset-act (T), 28
reset-transit (M), 21, 22
restart (R), 28, 29, 37, 38
restart-act (T), 28, 36, 37

save-contentbuffer (M), 47
search-member (M), 15, 31, 67
send-info (T), 33, 34, 38, 60, 63, 65, 66, 69
send-mail (M), 61, 62
send-mail (T), 68, 69
send-news (M), 61
send-news (T), 69
send-via-mail (T), 60–63, 68
send-via-news (T), 61, 64, 68
set-agent (M), 31, 32
set-agent-from-headers (M), 67
set-attribute (M), 11, 12
set-documentroot (M), 48
set-header (M), 66
set-ref (M), 12, 31, 67
setold (M), 50
setold (R), 50
show-agents (M), 30
show-doc-parts (M), 47
show-doc-parts (T), 48
simple-act (T), 27, 28, 37, 65
simple-doc (T), 46–48, 51–54, 57, 60, 63, 64
start (M), 22, 23
start-edit-doc (M), 42, 47, 48, 55
start-edit-doc (R), 43, 44, 49
start-edit-doc (T), 49
start-fin-state (T), 19, 22–25, 27, 28, 41, 42
State (A), 20
state-collection (T), 19, 22, 23, 28, 29
stop-edit-doc (M), 42, 47, 55
stop-edit-doc (R), 43, 44, 49
store-doc (M), 46, 51, 52
struct-activity (T), 28, 29, 31
struct-doc (T), 46–49, 51, 52, 54, 55
struct-rooted-doc (T), 46, 48, 54, 57
subtracttime (M), 18, 19

tex-doc (T), 54, 56–58
tex-dvi (T), 54, 56, 57, 64
tex-source (T), 54, 56, 57, 64

text-doc (T), 51, 53, 54, 57, 63
Time (A), 18
timed-collection (T), 16, 18, 19, 22, 23, 28, 29
timed-state (T), 19, 23, 28, 69
trans-by-mail (T), 65, 67–69
trans-by-news (T), 68, 69
trans-doc (T), 65, 66, 69
trans-info (T), 32, 33, 65, 68
transit-run (M), 21, 24, 25, 42
transit-step (M), 21, 24, 25, 42, 49
two-agent-trans (T), 33, 34, 37, 60, 63, 66–69
typeB (T), 8
typeC (T), 8
typeD (T), 8

undefined (S), 42–44, 49, 52, 55, 66
undo (R), 28, 29
undo-act (T), 28
unfinish (M), 22, 23
unstart (M), 22, 23

version-doc (T), 41, 44–46, 49–51
Versionhistory (A), 44
Versionnames (A), 45

with-descr (T), 12, 28, 33, 44
with-doc (T), 64, 65
with-headers (T), 63, 66–68
with-local-doc (T), 65, 66
with-newsgroup-header (T), 66–69
with-own-doc (T), 69
with-partner (T), 30, 32, 33, 36
with-state (T), 19–22, 24, 42
with-subject-header (T), 66, 67, 69
with-time (T), 16, 18, 19, 23